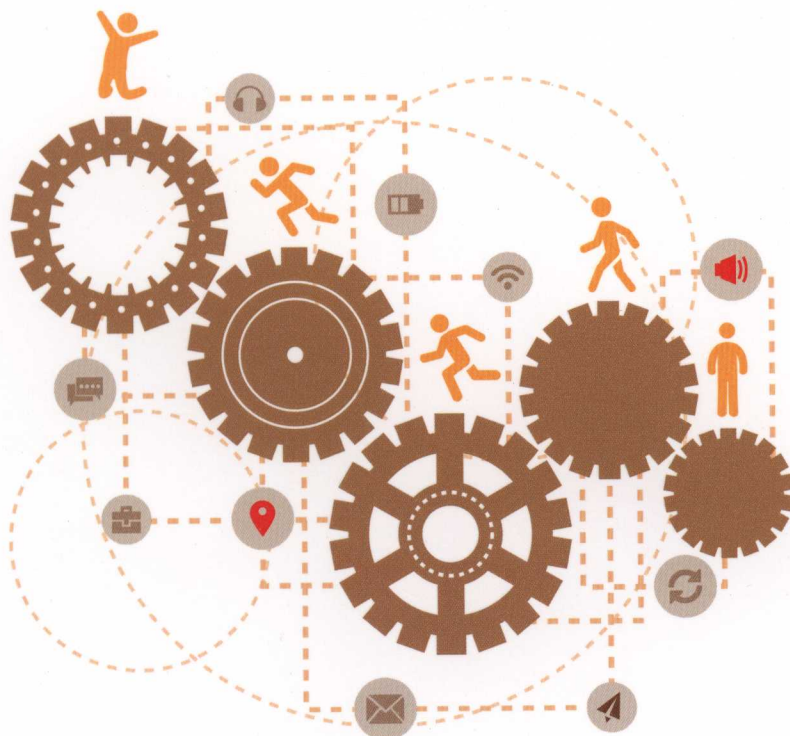


版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！



架构探险 从零开始写分布式服务框架

一线技术专家全方位解析分布式服务框架底层技术细节

李业兵 / 著

关于作者

李业兵



先后在支付宝运营支撑事业部、去哪儿网酒店事业部任职高级开发工程师。于2015年加入猫眼电影（原美团网旗下电影事业部），负责交易业务线架构与开发工作。

擅长电商交易领域系统设计与架构，在电商交易领域系统架构设计方面积累了较为丰富的实践经验。

对高并发系统设计、服务化架构、互联网中间件开发保持着浓厚的兴趣。

邮箱：liyebing1997@163.com



北京·BEIJING

内 容 简 介

本书的初衷是希望把分布式服务框架的实现细节及分布式服务框架周边的知识点梳理清楚,为那些对分布式服务框架感兴趣的人打开一扇窗户,降低获取相关知识的门槛。所以本书围绕实现分布式服务框架所需的知识点,进行了比较详尽细致的介绍。包括常见的RPC框架、常见的序列化/反序列化方案及选型、分布式服务框架服务的发布引入实现细节、软负载实现、底层通信方案实现、服务注册与发现实现、服务治理常见的功能等。通过对这些知识点的逐步讲解,层层深入,最终完成一个可运行的分布式服务框架。

通过这本书,读者可以完整地了解实现一个分布式服务框架的所有技术细节和实现原理,希望对想了解分布式服务框架实现细节的读者有所启发和帮助。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

架构探险:从零开始写分布式服务框架/李业兵著. —北京:电子工业出版社, 2017.7
ISBN 978-7-121-31959-4

I. ①架… II. ①李… III. ①分布式计算机系统—研究 IV. ①TP338.8

中国版本图书馆CIP数据核字(2017)第139692号

责任编辑:董英

印刷:三河市双峰印刷有限公司

装订:三河市双峰印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编100036

开本:787×980 1/16 印张:25 字数:365千字

版次:2017年7月第1版

印次:2017年7月第1次印刷

印数:3000册 定价:79.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:(010) 51260888-819, faq@phei.com.cn。

推荐序：做架构犹如去探险

2015 年，我写了自己的第一本书《架构探险：从零开始写 Java Web 框架》，书中记录了一款轻量级 Java Web 框架的整个开发过程。我将该框架取名为 Smart Framework，因为我认为 Java Web 框架应该更容易上手，运行得更快，更加轻量级。

为何我要写“架构探险”？

我希望有更多看过这本书的朋友，能够对 Java Web 框架的底层原理更加清楚，不仅能熟练使用框架，而且还能知道框架底层都做了些什么，做到“知其然，知其所以然”，我认为所有的开发者都应该这样要求自己。

此外，我认为学习技术的最有效方式就是分享，只有分享才能让自己得到更快的进步。但在分享之前，我们需要花时间去不断学习，这个学习过程也许是痛苦的，但当自己将所学技能分享出去的时候却是快乐的。不仅如此，我们通过分享还会收到大家更多的反馈，这些反馈能够帮助我们得到更加全面的成长。

架构和“探险”有何关系？

架构说简单点就是一堆技术、框架、工具的组合，至于怎么组合，这就非常考验架构师的经验和水平。一个优秀的架构，可以让开发效率变得更加高效，为企业节省更多的成本。程序员可将自己更多的精力放在业务需求的实现上，不会被底层的复杂技术细节所干扰。

架构师（或从事架构工作的人）就像是探险者一样，他们走在团队的前面，为大家铺路，带领大家找到成功捷径。此外，做架构工作不能照搬别人所谓的最佳实践，而要根据自身实际情况，因地制宜地灵活选择，设计最为合理的架构。架构的目的是为了让业务变得更加容易落地，降低开发成本与统一开发规范。架构师的职责就是避免大家踩坑，他们需要将自己的经验总结下来，并带领大家走最正确的路。架构师不只是在体验探险的凶险和快乐，而是把探险所积累的经验沉淀下来，让后面更多的人从中受益。

架构探险是一种信仰

其实我们都在架构中探险，或许自己目前正身陷险境，我们希望得到一本“宝典”，教会自己应该如何找到架构的成功捷径。李业兵老师写的这本《架构探险：从零开始写分布式服务框架》就能顺利地带领我们走出险境，让我们对分布式服务框架所涉及的技术了然于胸，并能合理地利用这些技术，搭建符合自身需求的分布式服务框架。

这本书传递了“架构探险”的精神，让我更加深刻地感受到，其实架构探险是一种信仰，它指引我们在架构之路上继续前进。

黄勇·特赞科技 CTO

2017年6月

前言

缘起

随着互联网浪潮风起云涌，互联网行业发展非常迅猛。此时将所有业务集中实现在一个应用上的做法已经满足不了公司及业务发展的需要了。基于面向服务体系架构来构建系统成了互联网架构师构建系统的不二选择，而面向服务体系架构能够落地的基础技术之一就是分布式服务框架。

要完全掌握分布式服务框架存在一定的技术门槛，市面上不乏一些非常出色的开源分布式服务框架。但对于新手而言，直接通过阅读源码来掌握分布式服务框架原理也并不是一件特别容易的事。

现在市面上也有专门的书籍来介绍分布式服务框架，但大都是从理论和方法论的角度来描述其原理的。有句俗语：“有些事，难不难，做了才知道；有些山，陡不陡，爬了才知道。”对于程序员来说，总是想通过具体的代码来了解一个分布式服务框架的实现细节，这样更为直观和深刻。为了帮助部分想了解分布式服务框架内部实现原理，甚至想自己实

现一个分布式服务框架的读者，我写了这样一本介绍如何从零开始写一个分布式服务框架的书，希望能够对想了解分布式服务框架实现细节的读者有所帮助。

内容大纲

全书一共 8 章。按照分布式服务框架的各个组成部分及各个组成部分所需的知识点或者这些知识点适当的延伸来组织每一章节的内容。建议读者按照全书章节的组织顺序来阅读。

第 1 章主要介绍日常开发常用的 RPC 框架，包括 RMI、CXF、Axis2、Thrift、gRPC、HttpClient，并就每一种 RPC 框架给出了实际可运行的代码示例，以及自己实现的一个简易版的 RPC 框架。

第 2 章对于基于服务体系架构做了介绍，对分布式服务框架总体架构及实现分布式服务框架所需的技术做了概要性介绍。

第 3 章介绍 9 种序列化/反序列化方案，每一种序列化/反序列化方案均给出了相应的代码示例，并给出了具体的选型建议。同时，将这 9 种序列化/反序列化实现集成在一起，实现了可配置化的序列化/反序列化工具引擎，最终整合在分布式服务框架实现内部。

第 4 章对 Spring 做了概要性介绍。有针对性地对 FactoryBean 周边知识及如何使用 FactoryBean 实现分布式服务的发布和引入做了详细介绍并给出代码实现。

第 5 章介绍 ZooKeeper 常用知识及如何使用 ZooKeeper 实现服务的注册与发现，并给出了具体的代码实现。

第 6 章围绕系统之间底层通信相关的知识点来组织，从 Java I/O 体系（阻塞 I/O、NIO、NIO2）到 Netty 相关知识均做了详细介绍。最后就使用 Netty 实现分布式服务框架底层通信给出了代码实现。

第 7 章介绍常用的软负载算法，并针对每一种算法给出了代码实现。同时将实现的多

种软负载算法集成可配置的软负载工具引擎，最终整合在分布式服务框架实现内部。

第8章介绍分布式服务框架服务治理相关的概念及方法论，并就部分服务治理功能给出了具体实现。

全书完整地实现了一个可以实际运行的分布式服务框架，全书所有代码均提供下载。

致谢

首先感谢我的妻子，在写书这段时间，宝宝的出生给了我人生中最好的礼物，宝宝的咿呀学语、一个不经意的笑容都能给我莫大的支持和鼓励。

同时，感谢猫眼电影公司的同事和领导，给了我宽松的学习与工作氛围，学到了很多知识，也得到了很多成长的机会。

最后，感谢辛苦劳作的编辑，本书能够出版有你们很大的一份功劳。

写在最后

这本书偏向实战，会有很多代码实现细节的描述，全书完整的代码实现会另给下载链接。本书所实现的分布式服务框架并未经历严苛生产环境的考验，定有很多不足之处，希望日后有机会再继续完善。写书对我来说是一个比较大的挑战，因为一个技术点，自己能理解和会用文字表达出来让别人也能理解是完全不同的层次。心中特别忐忑，担心因为自己对知识理解不够深入，以及文字表达水平不够，导致对读者有所误导。书中难免有错误和疏漏之处，在此恳请读者批评指正。

李业兵

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- ◎ **下载资源：**本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- ◎ **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- ◎ **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31959>



目录

第 1 章 常用的 RPC 框架	1
1.1 RPC 框架原理	1
1.2 RMI 介绍	2
1.2.1 原生 RMI 代码示例	3
1.2.2 RMI 穿透防火墙	5
1.3 CXF/Axis2 介绍	7
1.3.1 CXF 介绍	7
1.3.2 Axis2 介绍	14
1.4 Thrift 介绍	21
1.4.1 Thrift 工作原理介绍	23
1.4.2 Thrift IDL 语法说明	26
1.4.3 基于 Apache Thrift 的 Java 版完整案例	28
1.4.4 基于 Java 注解的简化实现	36
1.5 gRPC 介绍	42
1.5.1 protobuf3 语法介绍	43
1.5.2 gRPC 使用示例	45

1.6	HTTP Client 介绍	53
1.6.1	构建 HttpClient 对象	54
1.6.2	构建 URI 对象	55
1.6.3	构建请求对象 (HttpGet、HttpPost)	56
1.6.4	HttpClient 发起调用及获取调用返回结果	56
1.7	实现自己的 RPC 框架	61
1.8	RPC 框架与分布式服务框架的区别	68
1.9	本章小结	68
第 2 章	分布式服务框架总体架构与功能	69
2.1	面向服务的体系架构 (SOA)	69
2.1.1	面向服务架构范式	69
2.1.2	服务拆分原则	71
2.2	分布式服务框架现实需求	72
2.3	分布式服务框架总体架构及所需的技术概述	72
2.4	本章小结	74
第 3 章	分布式服务框架序列化与反序列化实现	75
3.1	序列化原理及常用的序列化介绍	75
3.2	Java 默认的序列化	77
3.3	XML 序列化框架介绍	80
3.4	JSON 序列化框架介绍	82
3.5	Hessian 序列化框架介绍	87
3.6	protobuf 序列化框架介绍	88
3.7	protostuff 序列化框架介绍	93
3.8	Thrift 序列化框架介绍	98
3.9	Avro 序列化框架介绍	100
3.9.1	Avro 介绍	100
3.9.2	Avro IDL 语言介绍	101

3.9.3	Schema 定义介绍	103
3.9.4	Maven 配置及使用 IDL 与 Schema 自动生成代码	103
3.9.5	Avro 序列化/反序列化实现	105
3.10	JBoss Marshalling 序列化框架介绍	110
3.11	序列化框架的选型	112
3.12	实现自己的序列化工具引擎	113
3.13	本章小结	118
第 4 章	实现分布式服务框架服务的发布与引入	119
4.1	Spring Framework 框架概述	119
4.1.1	Spring Framework 介绍	119
4.1.2	Spring Framework 周边生态项目介绍	121
4.2	FactoryBean 的秘密	122
4.2.1	FactoryBean 的作用及使用场景	123
4.2.2	FactoryBean 实现原理及示例说明	124
4.3	Spring 框架对于已有 RPC 框架集成的支持	127
4.3.1	Spring 支持集成 RPC 框架介绍	127
4.3.2	基于 RmiProxyFactoryBean 实现 RMI 与 Spring 的集成	128
4.3.3	基于 HttpInvokerProxyFactoryBean 实现 HTTP Invoker 与 Spring 的集成	131
4.3.4	基于 HessianProxyFactoryBean 实现 Hessian 与 Spring 的集成	133
4.4	实现自定义服务框架与 Spring 的集成	136
4.4.1	实现远程服务的发布	136
4.4.2	实现远程服务的引入	144
4.5	在 Spring 中定制自己的 XML 标签	150
4.6	本章小结	158
第 5 章	分布式服务框架注册中心	159
5.1	服务注册中心介绍	159
5.2	ZooKeeper 实现服务的注册中心原理	161

5.2.1	ZooKeeper 介绍	161
5.2.2	部署 ZooKeeper	161
5.2.3	ZkClient 使用介绍	164
5.2.4	ZooKeeper 实现服务注册中心	173
5.3	集成 ZooKeeper 实现自己的服务注册与发现	175
5.3.1	服务注册中心服务提供方	175
5.3.2	服务注册中心服务消费方	176
5.3.3	服务注册中心实现	178
5.4	本章小结	189
第 6 章	分布式服务框架底层通信实现	190
6.1	Java I/O 模型及 I/O 类库的进化	190
6.1.1	Linux 下实现的 I/O 模型	190
6.1.2	Java 语言实现的 I/O 模型	194
6.1.3	Java Classic I/O (Blocking I/O) 介绍	194
6.1.4	Java Non-blocking I/O (NIO) 介绍	211
6.1.5	NIO2 及 Asynchronous I/O 介绍	233
6.2	Netty 使用介绍	255
6.2.1	Netty 开发入门	256
6.2.2	Netty 粘包/半包问题解决	265
6.3	使用 Netty 构建服务框架底层通信	320
6.3.1	构建分布式服务框架 Netty 服务端	320
6.3.2	构建分布式服务框架服务调用端 Netty 客户端	330
6.4	本章小结	347
第 7 章	分布式服务框架软负载实现	348
7.1	软负载的实现原理	348
7.2	负载均衡常用算法	349
7.2.1	软负载随机算法实现	349

7.2.2	软负载加权随机算法实现	350
7.2.3	软负载轮询算法实现	351
7.2.4	软负载加权轮询算法实现	352
7.2.5	软负载源地址 hash 算法实现	354
7.3	实现自己的软负载机制	355
7.4	软负载在分布式服务框架中的应用	357
7.5	本章小结	361
第 8 章	分布式服务框架服务治理	362
8.1	服务治理介绍	362
8.2	服务治理的简单实现	364
8.2.1	服务分组路由实现	364
8.2.2	简单服务依赖关系分析实现	374
8.2.3	服务调用链路跟踪实现原理	380
8.3	本章小结	380
附录 A	如何配置运行本书完成的分布式服务框架	381

第 1 章

常用的 RPC 框架

1.1 RPC 框架原理

RPC (Remote Procedure Call, 远程过程调用) 一般用来实现部署在不同机器上的系统之间的方法调用, 使得程序能够像访问本地系统资源一样, 通过网络传输去访问远端系统资源。RPC 框架实现的架构原理都是类似的, 如图 1-1 所示。

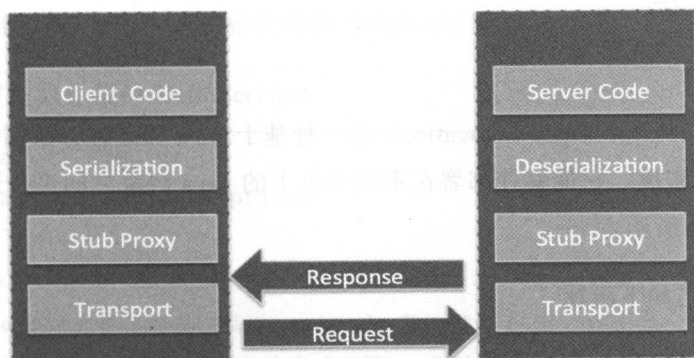


图 1-1 RPC 框架原理

在该架构中，下面几个方面是重点。

- ◎ **Client Code**: 客户端调用方代码实现，负责发起 RPC 调用，为调用方用户提供使用 API。
- ◎ **Serialization/Deserialization**: 负责对 RPC 调用通过网络传输的内容进行序列化与反序列化，不同的 RPC 框架有不同的实现机制。主要分为文本与二进制两大类。文本类别的序列化机制主要有 XML 与 JSON 两种格式，二进制类别的序列化机制常见的有 Java 原生的序列化机制，以及 Hessian、protobuf、Thrift、Avro、Kryo、MessagePack 等，不同的序列化方式在可读性、码流大小、支持的数据类型及性能等方面都存在较大差异，需要用户根据自己的实际情况进行甄别与筛选。
- ◎ **Stub Proxy**: 可以看作是一种代理对象，屏蔽 RPC 调用过程中复杂的网络处理逻辑，使得 RPC 调用透明化，能够保持与本地调用一样的代码风格。
- ◎ **Transport**: 作为 RPC 框架底层的通信传输模块，一般通过 Socket 在客户端与服务端之间传递请求与应答消息。
- ◎ **Server Code**: 服务端服务业务逻辑具体的实现。

下面几节分别针对目前常用的几类 RPC 框架做介绍。

1.2 RMI 介绍

Java RMI (Remote Method Invocation) 是一种基于 Java 的远程方法调用技术，是 Java 特有的一种 RPC 实现。它能够使部署在不同主机上的 Java 对象之间进行透明的通信与方法调用，如图 1-2 所示。

RMI 特性总结如下所述。

- ◎ 支持真正的面向对象的多态性。而完全支持面向对象也是 RMI 相对于其他 RPC 框架的优势之一。

- ◎ Java 语言独有，不支持其他语言，能够完美地支持 Java 语言所独有的特性。
- ◎ 使用了 Java 原生的序列化机制，所有序列化对象必须实现 `java.io.Serializable` 接口。
- ◎ 底层通信基于 BIO（同步阻塞 I/O）实现的 Socket 完成。

因 Java 原生序列化机制与 BIO 通信机制本身存在性能问题，导致 RMI 的性能较差，对性能要求高的使用场景不推荐该方案。

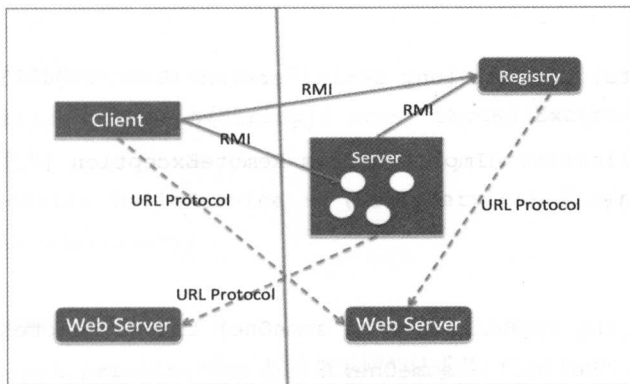


图 1-2 RMI 调用示意图

1.2.1 原生 RMI 代码示例

RMI 使用示例如下所述（本书相关代码可到本书资源页面下载）。

定义 RMI 对外服务接口 `HelloService`：

```

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloService extends Remote {
    String sayHello(String someOne) throws RemoteException;
}
  
```

RMI 接口方法定义必须显式声明抛出 `RemoteException` 异常。

服务端接口实现 `HelloServiceImpl`:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloServiceImpl extends UnicastRemoteObject implements
HelloService {

    private static final long serialVersionUID = -6190513770400890033L;

    public HelloServiceImpl() throws RemoteException {
        super();
    }
    @Override
    public String sayHello(String someOne) throws RemoteException {
        return "Hello," + someOne;
    }
}
```

服务端方法实现必须继承 `UnicastRemoteObject` 类，该类定义了服务调用方与服务提供方对象实例，并建立一对一的连接。

服务端 RMI 服务启动代码:

```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class ServerMain {
    public static void main(String[] args) throws Exception {
        //创建服务
        HelloService helloService = new HelloServiceImpl();
        //注册服务
        LocateRegistry.createRegistry(8801);
    }
}
```

```

        Naming.bind("rmi://localhost:8801/helloService", helloService);
        System.out.println("ServerMain provide RPC service now");
    }
}

```

客户端远程调用 RMI 服务代码:

```

import rmi.study.server.HelloService;
import java.rmi.Naming;

public class ClientMain {
    public static void main(String[] args) throws Exception {
        //服务引入
        HelloService helloService = (HelloService) Naming.lookup("rmi://localhost:8801/helloService");

        //调用远程方法
        System.out.println("RMI 服务器返回的结果是: " + helloService.sayHello("liyebing"));
    }
}

```

首先运行服务端程序 **ServerMain**，然后运行 **ClientMain**，运行结果如下:

```
RMI 服务器返回的结果是: Hello, liyebing
```

1.2.2 RMI 穿透防火墙

RMI 的通信端口是随机产生的，因此有可能会被防火墙拦截。为了防止被防火墙拦截，需要强制指定 RMI 的通信端口。一般通过自定义一个 **RMI SocketFactory** 类来实现，代码如下:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.rmi.server.RMISocketFactory;

public class CustomerSocketFactory extends RMISocketFactory {
    //指定通信端口，防止被防火墙拦截
    @Override
    public Socket createSocket(String host, int port) throws IOException {
        return new Socket(host, port);
    }

    @Override
    public ServerSocket createServerSocket(int port) throws IOException {
        if (port == 0) {
            port = 8501;
        }
        System.out.println("rmi notify port:" + port);
        return new ServerSocket(port);
    }
}
```

然后将新定义类注入到 Rmiserver 端，代码如下：

```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.RMISocketFactory;

public class ServiceMain {
    public static void main(String[] args) throws Exception {
        LocateRegistry.createRegistry(8801);
        //指定通信端口，防止被防火墙拦截
        RMISocketFactory.setSocketFactory(new CustomerSocketFactory());
    }
}
```

```

HelloService helloService = new HelloServiceImpl();
Naming.bind("rmi://localhost:8801/helloService",helloService);
System.out.println("ServiceMain provide RPC service now.");
}
}

```

1.3 CXF/Axis2 介绍

WebService 是一种跨平台的 RPC 技术协议。WebService 技术栈由 SOAP (Sample Object Access Protocol, 简易对象访问协议)、UDDI (Universal Description, Discovery and Integration, 统一描述、发现与集成)、WSDL (Web Services Description Language, 网络服务描述语言) 组成。其中, SOAP 是一种使用 XML 进行数据编码的通信协议, 独立于平台, 独立于语言, 简单可扩展, 因为 SOAP 基于 HTTP 协议进行数据传输, 故能绕过防火墙。SOAP 提供了一种标准方法, 使得运行在不同的操作系统并使用不同技术和编程语言的应用程序可以互相通信。UDDI 是一个独立于平台的框架, 是一种通用描述、发现与集成服务。WSDL 是使用 XML 编写的网络服务描述语言, 用来描述 WebService, 以及如何访问 WebService。

下面介绍 Java 开发中常用的两种 WebService 实现: CXF 与 Axis2。

1.3.1 CXF 介绍

Apache CXF 是一个开源的 WebService RPC 框架, 是两个著名的开源项目 ObjectWeb Celtix 和 Codehaus XFire 合并后的产物。Apache CXF 包含一个范围广泛、功能齐全的集合。主要有以下特点。

- ◎ 支持 Web Services 标准, 包括 SOAP (SOAP1.1 和 SOAP1.2) 规范, WSI Basic Profile、WSDL、WS-Addressing、WS-Policy、WS-ReliableMessageing、WS-Security 等。

- ◎ 支持 JSR 相关规范和标准，包括 JAX-WS (Java API for XML-Based Web Services 2.0)、JAX-RS (The Java API for RESTful Web Services)、SAAJ (SOAP with Attachments API for Java)。
- ◎ 支持多种传输协议和协议绑定、数据绑定。
- 协议绑定：SOAP、REST/HTTP、XML。
- 数据绑定：JAXB 2.X、Aegis、Apache XMLBeans 等。

CXF 官方网站是 cxf.apache.org。

下面通过一个集成 Spring 基于 Maven 的 Java Web 项目来说明 CXF 的用法。为了演示方便，将 WebService 的服务端与客户端代码写到了一个工程里面。工程的结构如图 1-3 所示。

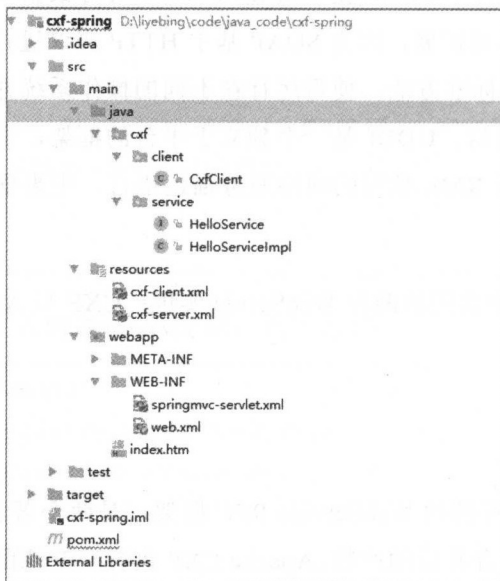


图 1-3 CXF 演示项目工程结构

CXF 集成 Spring Maven 项目示例关键代码如下所述。

定义服务端服务接口 `HelloService`，其中接口使用注解 `@WebService` 标明是一个

WebService 远程服务接口:

```
package cxf.service;
import javax.ws.WebService;

@WebService
public interface HelloService {
    public String sayHello(String content);
}

package cxf.service;
import javax.ws.WebService;
@WebService(endpointInterface = "cxf.service.HelloService")
public class HelloServiceImpl implements HelloService {
    public String sayHello(String content) {
        return "hello," + content;
    }
}
```

上述代码中 WebService 服务接口的实现类为 HelloServiceImpl, 同时使用注解 @WebService, 并通过 endpointInterface 指明对应的接口实现 cxf.service.HelloService。

cxf-server.xml 配置的主体内容如下:

```
<import resource="classpath:META-INF/cxf/cxf.xml"/>
<import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>
<jaxws:endpoint id="helloWorld"
implementor="cxf.service.HelloServiceImpl" address="/HelloWorld"/>
```

web.xml 配置如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
```

```
        id="WebApp_ID" version="2.5">
<display-name>cxfr-spring</display-name>
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:cxfr-server.xml</param-value>
</context-param>
<listener>
<listener-class>
        org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
<servlet>
<servlet-name>CXFServlet</servlet-name>
<servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>CXFServlet</servlet-name>
<url-pattern>/ws/*</url-pattern>
</servlet-mapping>

</web-app>
```

pom.xml Maven 依赖配置如下：

```
<modelVersion>4.0.0</modelVersion>
<groupId>cxfr-spring-study</groupId>
<artifactId>cxfr-spring</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>
<properties>
<cxfr.version>3.1.7</cxfr.version>
<spring.version>4.0.9.RELEASE</spring.version>
</properties>
```



```
<dependencies>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>${spring.version}</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>${spring.version}</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context-support</artifactId>
<version>${spring.version}</version>
</dependency>
<dependency>
<groupId>org.apache.cxf</groupId>
<artifactId>cxf-rt-frontend-jaxws</artifactId>
<version>${cxf.version}</version>
</dependency>
<dependency>
<groupId>org.apache.cxf</groupId>
<artifactId>cxf-rt-transports-http</artifactId>
<version>${cxf.version}</version>
</dependency>
<dependency>
<groupId>org.apache.cxf</groupId>
<artifactId>cxf-rt-transports-http-jetty</artifactId>
<version>${cxf.version}</version>
</dependency>
</dependencies>
```

在 Tomcat Web 容器中部署运行，结果如图 1-4 所示。

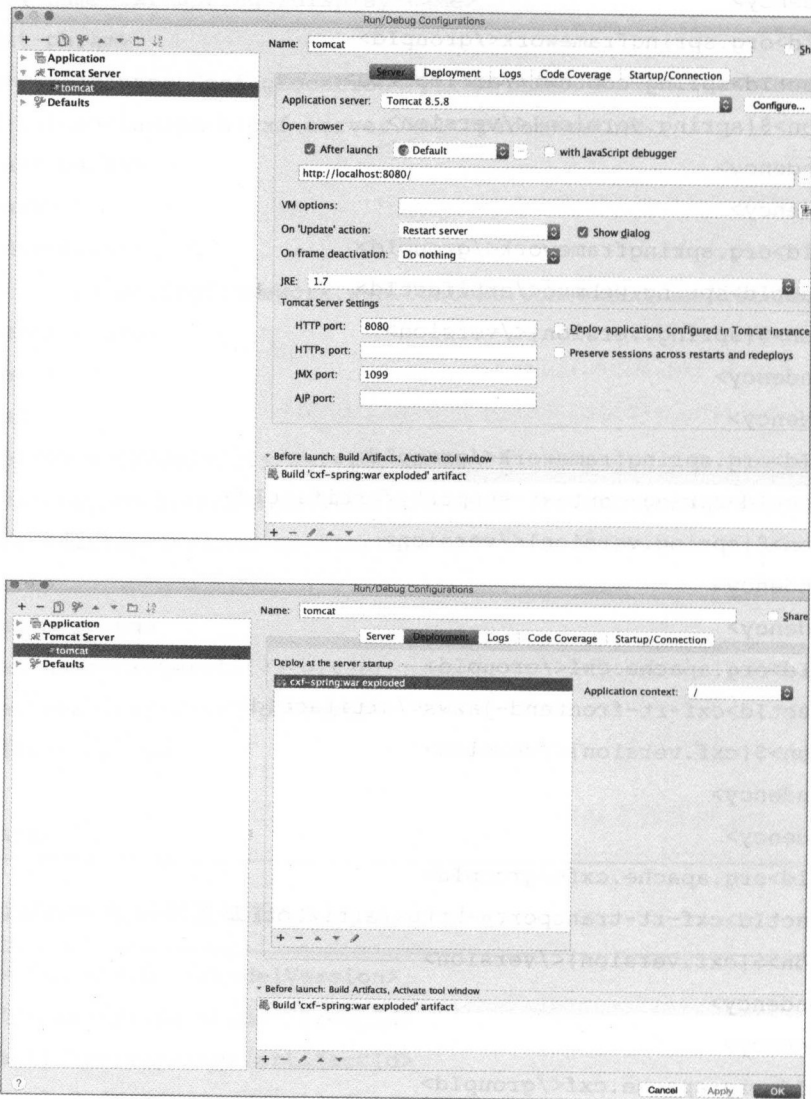


图 1-4 在 Tomcat 中部署运行

然后在浏览器中运行 localhost:8080/ws/HelloWorld?wsdl，得到如图 1-5 所示的结果，说明 WebService 服务端服务发布完成。



图 1-5 CXF WebService 服务发布 WSDL 文件

下面介绍客户端如何调用服务端发布的 WebService 服务。

首先配置服务的引入信息 cxf-client.xml，主体内容如下：

```
<jaxws:client id="helloClient"
              serviceClass="cxf.service.HelloService"
              address="http://localhost:8080/ws/HelloWorld" />
</beans>
```

其中 serviceClass 配置服务端发布的 WebService 服务接口，address 为服务端服务访问 HTTP 地址（WebService 采用 HTTP 协议通信）。

最后，编写客户端调用类 CxfClient：

```
package cxf.client;

import cxf.service.HelloService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class CxfClient {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext  
("classpath:cxf-client.xml");  
        HelloService client = (HelloService) context.getBean("helloClient");  
        System.out.println(client.sayHello("liyebing"));  
    }  
}
```

运行结果如下：

hello,liyebing

1.3.2 Axis2 介绍

Apache Axis2 是 Axis 的后续版本，是新一代的 SOAP 引擎，是 CXF 之外另一个非常流行的 Web Services/SOAP/WSDL 实现，且同时存在 Java 语言与 C 语言两种实现。这里就 Java 语言的实现做一个介绍，参考官网 <http://axis.apache.org/axis2/java/core/>。

Axis2 主要有以下特点。

- ◎ 高性能：Axis2 具有自己的轻量级对象模型 AXIOM，且采用 StAX (Streaming API for XML) 技术，具有更优秀的性能表现，比 Axis1.x 的内存消耗更低。
- ◎ 热部署：Axis2 配备了在系统启动和运行时部署 Web 服务和处理程序的功能。换句话说，可以将新服务添加到系统，而不必关闭服务器。只需将所需的 Web 服务归档存储在存储库的 services 目录中，部署模型将自动部署该服务并使其可供使用。
- ◎ 异步服务支持：Axis2 支持使用非阻塞客户端和传输的异步，以及 Web 服务和异步 Web 服务调用。

- ◎ **WSDL 支持**: Axis2 支持 Web 服务描述语言版本 1.1 和 2.0, 它允许轻松构建存根以访问远程服务, 还可以自动导出来自 Axis2 的已部署服务的机器可读描述。

下面通过一个集成 Spring 基于 Maven 的 Java Web 项目来说明 Axis2 的用法。为了演示方便, 将 WebService 的服务端与客户端代码写到了一个工程里面。工程的结构如图 1-6 所示。

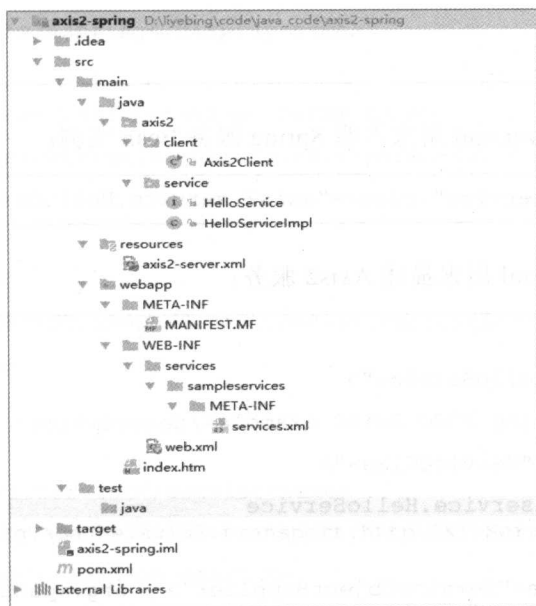


图 1-6 Axis2 集成 Spring Maven 工程结构

Axis2 Web Service 服务端关键代码如下所述。

定义 WebService 服务接口:

```
package axis2.service;

public interface HelloService {
    public String sayHello(String content);
}
```

服务接口 `HelloService` 实现：

```
package axis2.service;

public class HelloServiceImpl implements HelloService {

    public String sayHello(String content) {
        return "hello," + content;
    }
}
```

配置文件 `axis2-server.xml` 用来声明 Spring 服务 Bean 实例：

```
<bean id="helloService" class="axis2.service.HelloServiceImpl"/>
```

配置文件 `services.xml` 用来描述 Axis2 服务：

```
<serviceGroup>
<service name="HelloService">
<description>Spring POJO Axis2 example</description>
<parameter name="ServiceClass">
    axis2.service.HelloService
</parameter>
<parameter name="ServiceObjectSupplier"> org.apache.axis2.extensions.
spring.receivers.SpringServletContextObjectSupplier
</parameter>
<parameter name="SpringBeanName">helloService</parameter>
<messageReceivers>
<messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out" class="org.
apache.axis2.rpc.receivers.RPCMessageReceiver" />
</messageReceivers>
</service>
</serviceGroup>
```

其中 `web.xml` 内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">

    <display-name>axis2-spring</display-name>
    <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:axis2-server.xml</param-value>
    </context-param>
    <listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
    </listener>
    <servlet>
    <servlet-name>AxisServlet</servlet-name>
    <servlet-class>org.apache.axis2.transport.http.AxisServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
    </web-app>

```

配置文件 pom.xml Maven 依赖内容如下：

```

<modelVersion>4.0.0</modelVersion>
<groupId>axis2-spring-study</groupId>
<artifactId>axis2-spring</artifactId>
<version>1.0-SNAPSHOT</version>

```

```
<packaging>war</packaging>
<properties>
<axis2.version>1.7.4</axis2.version>
<spring.version>4.0.9.RELEASE</spring.version>
</properties>
<dependencies>
<!-- spring -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>${spring.version}</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>${spring.version}</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context-support</artifactId>
<version>${spring.version}</version>
</dependency>
<!-- axis2 -->
<dependency>
<groupId>org.apache.axis2</groupId>
<artifactId>axis2-transport-xmlpp</artifactId>
<version>${axis2.version}</version>
</dependency>
<dependency>
<groupId>org.apache.axis2</groupId>
<artifactId>axis2-transport-udp</artifactId>
<version>${axis2.version}</version>
</dependency>
```



```

<dependency>
<groupId>org.apache.axis2</groupId>
<artifactId>axis2-transport-tcp</artifactId>
<version>${axis2.version}</version>
</dependency>
<dependency>
<groupId>org.apache.axis2</groupId>
<artifactId>axis2-webapp</artifactId>
<version>${axis2.version}</version>
</dependency>
<dependency>
<groupId>org.apache.axis2</groupId>
<artifactId>axis2-xmlbeans</artifactId>
<version>${axis2.version}</version>
</dependency>
<dependency>
<groupId>org.apache.axis2</groupId>
<artifactId>axis2-spring</artifactId>
<version>${axis2.version}</version>
</dependency>
<dependency>
<groupId>org.apache.axis2</groupId>
<artifactId>axis2-soapmonitor-servlet</artifactId>
<version>${axis2.version}</version>
</dependency>
</dependencies>

```

在 Tomcat 中部署运行,然后在浏览器中运行 <http://localhost:8080/services/HelloService?wsdl>,得到如图 1-7 所示的内容,说明 WebService 服务端发布完成。



图 1-7 WebService 服务发布 WSDL 文件

Axis2 客户端调用的方式有多种，下面介绍一种不用预生成代码的动态调用方式。

```

import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.rpc.client.RPCServiceClient;
import org.apache.axis2.transport.http.HTTPConstants;
import javax.xml.namespace.QName;

public class Axis2Client {
    public static void main(String[] args) {
        try {
            EndpointReference targetEPR = new EndpointReference(
                "http://localhost:8080/services/HelloService");
            RPCServiceClient serviceClient = new RPCServiceClient();

            Options options = serviceClient.getOptions();
            options.setManageSession(true);
            options.setProperty(HTTPConstants.REUSE_HTTP_CLIENT, true);
            options.setTo(targetEPR);

```

```

        QName opQName = new QName("http://service.axis2", "sayHello");
        //设置调用参数
        Object[] paramArgs = new Object[]{"liyebing"};
        //处理返回数据
        Class[] returnTypes = new Class[]{String.class};
        Object[] response = serviceClient.invokeBlocking(opQName,
paramArgs, returnTypes);
        serviceClient.cleanupTransport();
        String result = (String) response[0];
        if (result == null) {
            System.out.println("HelloService didn't initialize!");
        } else {
            System.out.println(result);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

运行结果如下:

hello,liyebing

1.4 Thrift 介绍

Apache Thrift 是跨越不同的平台和语言, 协助构建可伸缩的分布式系统的一种 RPC 实现。最初由 Facebook 内部开发使用, 现在已经开源。它的特点是具备广泛的语言支持, 以及高性能。当前最新版本是 0.9.3, 已经支持如下主流编程语言:

- ◎ ActionScript3
- ◎ C++
- ◎ C#
- ◎ D
- ◎ Delphi
- ◎ Erlang
- ◎ Haskell
- ◎ Java
- ◎ JavaScript
- ◎ Node.js
- ◎ Objective-C/Cocoa
- ◎ OCaml
- ◎ Perl
- ◎ PHP
- ◎ Python
- ◎ Ruby
- ◎ Smalltalk

Apache Thrift 作为在多语言并存的异构系统之间的 RPC 调用方案是一个非常不错的选择，当然也可以作为同构系统之间的 RPC 方案。尤其对比 XML-RPC/JSON-RPC/SOAP 与 WSDL 协议栈实现的 RPC 方案，有着非常明显的性能优势。原因在于，Thrift 是采用二进制编码协议、使用 TCP/IP 传输协议的一种 RPC 实现，而 XML-RPC/JSON-RPC/SOAP

与 WSDL 协议栈采用文本协议，WSDL 的实现 WebService 采用 HTTP 作为传输协议。对于网络数据传输，TCP/IP 协议的性能要高于 HTTP 协议，不仅因为 HTTP 协议是应用层协议，HTTP 协议传输内容除应用数据本身外，还带有不少描述本次请求上下文的数据（比如响应状态码、Header 信息等）。此外，HTTP 协议一般使用文本协议对传输内容进行编码，相对于一般采用二进制编码协议的 TCP/IP 协议码流要大。

1.4.1 Thrift 工作原理介绍

从使用者的角度来看，Apache Thrift 工作流程如图 1-8 所示。

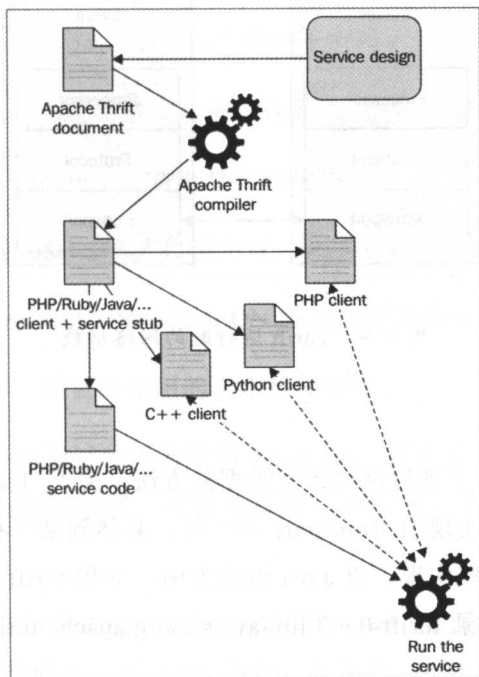


图 1-8 Apache Thrift 的工作流程

Thrift 的工作流程总结如下所述。

- ◎ 设计所需要的服务。

- ◎ 根据设计的服务，编写 Thrift IDL 服务描述文件。
- ◎ 根据编写的 Thrift IDL 服务描述文件使用 Thrift 提供的代码生成工具生成服务端与客户端的代码。
- ◎ 实现服务端业务逻辑的编写，同时实现客户端调用代码的编写。
- ◎ 运行服务端与客户端。

从 Thrift 内部运行的角度来看，Thrift 运行时的网络堆栈包括 Transport、Protocol、Processor 和 Server 四个部分，如图 1-9 所示。

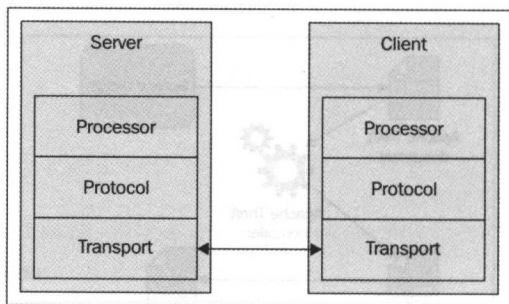


图 1-9 Thrift 运行时的网络堆栈

下面分别进行介绍。

(1) Transport: 提供了通过网络读写数据的方法。关于 Transport，文档方面比较缺失，而且不同的编程语言实现的 Transport 不一样。具体到某一种编程语言，需要自己通过实现的源码来分析有哪些实现。以 Java 语言为例，下载 thrift-0.9.3.tar.gz，Java 相关的 Transport 的实现请查看目录 thrift-0.9.3\lib\java\src\org\apache\thrift\transport。Java 语言主要的实现如下所述。

- ◎ TSocket 与 TIOStreamTransport 基于阻塞 I/O 模型实现，每次请求独占一个连接。效率比较低，不推荐在生产环境中使用。
- ◎ TNonblockingTransport、TNonblockingSocket 这两个类对应着非阻塞 I/O 实现。

- ◎ **TMemoryInputTransport** 封装了一个字节数组 `byte[]` 来做输入流的封装。
TMemoryBuffer 使用字节数组输出流 `ByteArrayOutputStream` 做输出流的封装。
- ◎ **TFramedTransport** 则封装了 **TMemoryInputTransport** 做输入流，封装 **TByteArrayOutPutStream** 做输出流，作为内存读写缓冲区的一个封装。
- ◎ **TFastFramedTransport** 是内存利用率更高的一个内存读写缓存区，它使用自动增长的 `byte[]`（长度不够才新建），而不是每次都新建一个 `byte[]`，提高了内存的使用率。

(2) **Protocol**: 提供了对网络传输数据进行序列化/反序列化的具体实现。具体到某种编程语言,可以通过实现的源码来分析有哪些实现。以Java为例,目录为 `thrift-0.9.3\lib\java\src\org\apache\thrift\protocol`。Java的常用实现如下所述。

- ◎ **TbinaryProtocol**: 二进制格式传输协议。
- ◎ **TCompactProtocol**: 压缩二进制格式传输协议。
- ◎ **TJSONProtocol**: JSON格式传输协议。
- ◎ **TSimpleJSONProtocol**: 简单的JSON格式数据传输协议。
- ◎ **TDebugProtocol**: 调试时使用的文本传输协议。

(3) **Processor**: Thrift 通过使用编写的 Thrift IDL 描述文件来自动生成 **Processor**。它从负责输入的 **Protocol** 读取数据,将其传递给处理程序,并将结果发送到负责输出的 **Protocol**。

(4) **Server**: **Server** 将 **Transport**、**Protocol**、**Processor** 组合在一起,将服务运行起来,在指定端口等待调用端的请求。以Java为例,目录为 `thrift-0.9.3\lib\java\src\org\apache\thrift\server`。常用的实现如下所述。

- ◎ **TnonblockingServer**: 基于多线程非阻塞 I/O 模型实现,适用于连接数较多的高并发环境。

- ◎ **TThreadPoolServer**: 基于多线程阻塞 I/O 模型实现, 比 **TNonblockingServer** 需要耗费更多的系统资源。
- ◎ **TThsHaServer**: 半同步、半异步服务器。
- ◎ **TsimpleServer**: 基于单线程的阻塞 I/O 模型实现, 主要用于测试, 不推荐在生产环境中使用。

1.4.2 Thrift IDL 语法说明

1. Thrift 常用的数据类型

(1) 基本类型。

- ◎ **bool**: 布尔值 true/false。
- ◎ **byte**: Byte 类型或者 8 位有符号整数。
- ◎ **i16**: 16 位有符号整数。
- ◎ **i32**: 32 位有符号整数。
- ◎ **i64**: 64 位有符号整数。
- ◎ **double**: 64 位双精度符号浮点数。
- ◎ **string**: UTF-8 格式编码的字符串。

(2) 容器类型。

- ◎ **list<type>**: 数组数据类型。
- ◎ **set<type>**: 集合数据类型, 不同于数组, 集合内部的元素保持唯一。
- ◎ **map<type1,type2>**: 对应于键值对 (Key-Value) 数据结构 Map, 其中 Key 保持唯一性。

(3) 结构体类型。

struct: 一组强类型对象的封装，类似于C语言中的 **struct** 类型。

例如：

```
struct User{
    1:string name;
    2:i32 age;
}
```

(4) enum 枚举类型。

和 Java 语言中的枚举类型含义一致。Thrift 不支持枚举类嵌套，枚举常量必须是 32 位的正整数。举例如下：

```
enum Status {
    ONE,
    TWO = 2,
    THREE = 3
}
```

2. 声明服务

声明服务的语法类似面向对象编程里面的接口的写法。服务由一组函数声明组成，每个函数都有参数、返回类型和关于抛出异常的可选信息。也可以将函数声明为 **oneway**，这种情况下调用端不会等待该服务的响应（此时，该函数的返回类型必须是 **void**）。

举例如下：

```
namespace java thrift.gencode.server
service HelloService{
    string sayHello(1:UserModel.User user,2:string content);
    oneway void notifyMessage();
}
```

其中 `namespace` 是 Thrift 的统一命名空间，通过设置命名空间，可以指定 Thrift 生成文件的位置。具体到 Java 语言，则是通过 `namespace` 指定生成 Java 文件的包路径。

3. 服务升级保持兼容性

已经写好的服务随着时间的变化可能不再符合当初的设计要求，需要做变更，比如需要新增字段。如果需要与变更之前 Thrift 文件产生的代码相兼容，需要注意以下两点。

(1) 不要变更之前已经存在的字段的编号值。

(2) 新添加的字段可以使用 `optional` 进行修饰，以便格式兼容。

例如，为 User 新增 `email` 字段，可以在 `struct User` 中按序号递增新增属性 `email`，同时使用 `optional` 修饰符，代表可选非必须属性：

```
struct User{  
    1:string name;  
    2:i32 age;  
    3:optional string email;  
}
```

下面以 Java 语言为例，编写一个完整的基于 Apache Thrift 的案例。

1.4.3 基于 Apache Thrift 的 Java 版完整案例

首先，编写 Thrift IDL 文件，这里一共有两个 Thrift IDL 文件，`UserModel.thrift` 定义了数据结构，`HelloService.thrift` 定义了服务的声明，关键代码如下所述。

数据结构定义 `UserModel.thrift`：

```
namespace java thrift.gencode.server  
  
struct User{  
    1:string name;  
    2:string email;  
}
```

服务接口定义 `HelloService.thrift`:

```
namespace java thrift.gencode.server
include 'UserModel.thrift'
service HelloService{
    string sayHello(1:UserModel.User user);
}
```

打开命令行 `cmd.exe`, 进入 `thrift-0.9.3.exe`[下载地址为 <http://thrift.apache.org/download>] 所在的目录, 在 Windows 命令行中分别执行下面命令:

```
thrift-0.9.3.exe -r -gen java UserModel.thrift
thrift-0.9.3.exe -r -gen java HelloService.thrift
```

会在 `thrift-0.9.3.exe` 所在的目录下生成如下文件:

```
Gen-java\thrift\gencode\HelloService.java
Gen-java\thrift\gencode\User.java
```

将文件复制到自己的工程目录里面。

基于上面所生成的文件编写服务端代码, 实现业务逻辑, 代码如下:

```
package thrift.gencode.server;

import org.apache.thrift.TException;

public class HelloServiceImpl implements HelloService.Iface {
    public String sayHello(User user) throws TException {
        return "hello," + user.getName() + user.getEmail();
    }
}
```

然后, 编写服务启动方法与服务调用方法, 完成服务的调用。

阻塞同步模式代码示例:

```
public class SimpleInvoker {

    /**
     * 启动服务
     */
    public void startServer() throws TTransportException {
        //创建 processor
        TProcessor tprocessor = new HelloService.Processor<HelloService.
        Iface>(new HelloServiceImpl());
        //服务端点
        int port = 8091;
        //创建 transport 阻塞通信
        TServerSocket serverTransport = new TServerSocket(port);
        //创建 protocol
        TBinaryProtocol.Factory protocol = new TBinaryProtocol.Factory();
        //将 processor transport protocol 设置到服务器 server 中
        TServer.Args args = new TServer.Args(serverTransport);
        args.processor(tprocessor);
        args.protocolFactory(protocol);
        //定义服务器类型设定参数
        TServer server = new TSimpleServer(args);
        //开启服务
        server.serve();
    }

    /**
     * 客户端调用服务端
     */
    public void startClient() throws TException {
        String ip = "127.0.0.1";
        int port = 8091;
        int timeOut = 1000;
```

```

//创建 Transport
TTransport transport = new TSocket(ip, port, timeOut);
//创建 TProtocol
TProtocol protocol = new TBinaryProtocol(transport);
//基于 TTransport 和 TProtocol 创建 Client
HelloService.Client client = new HelloService.Client(protocol);
transport.open();

//调用 client 方法
User user = new User();
user.setName("liyebing");
user.setEmail("test@163.com");
String content = client.sayHello(user);
System.out.println("content:" + content);
}
}

```

异步非阻塞模式代码示例:

```

/**
 * 非阻塞 I/O 服务调用示例
 */
public class NonblockingInvoker {

    /**
     * 启动服务
     */
    public void startServer() throws TTransportException {
        int port = 8091;
        //创建 processor
        TProcessor tprocessor = new HelloService.Processor<HelloService.
        Iface>(new HelloServiceImpl());
        //创建 transport 非阻塞 nonblocking
    }
}

```

```
TNonblockingServerTransport    serverTransport    =    new
TNonblockingServerSocket(port);
    //创建 protocol 数据传输协议
    TCompactProtocol.Factory protocol = new TCompactProtocol.Factory();
    //创建 transport 数据传输方式，非阻塞需要用这种方式传输
    TFramedTransport.Factory    transport    =    new    TFramedTransport.
Factory();
    TNonblockingServer.Args    args    =    new    TNonblockingServer.Args
(serverTransport);
    args.processor(tprocessor);
    args.transportFactory(transport);
    args.protocolFactory(protocol);
    //创建服务器，类型是非阻塞
    TServer server = new TNonblockingServer(args);
    //开启服务
    server.serve();
}

/**
 * 客户端调用服务端
 */
public void startClient() throws Exception {
    String ip = "127.0.0.1";
    int port = 8091;
    int timeOut = 1000;

    TAsyncClientManager clientManager = new TAsyncClientManager();
    TNonblockingTransport transport = new TNonblockingSocket(ip, port,
timeOut);
    TProtocolFactory tprotocol = new TCompactProtocol.Factory();
    HelloService.AsyncClient asyncClient = new HelloService.AsyncClient
(tprotocol, clientManager, transport);
```

```

//客户端异步调用
User user = new User();
user.setName("liyebing");
user.setEmail("test@163.com");
CountDownLatch latch = new CountDownLatch(1);
//设置回调接口实现
AsyncInvokerCallback callBack = new AsyncInvokerCallback(latch);
asyncClient.sayHello(user, callBack);
//等待调用返回
latch.await(5, TimeUnit.SECONDS);
}
}

```

其中, AsyncInvokerCallback 为异步调用回调结果类:

```

public class AsyncInvokerCallback implements AsyncMethodCallback
<HelloService.AsyncClient.sayHello_call> {

    private CountDownLatch latch;

    public AsyncInvokerCallback(CountDownLatch latch) {
        this.latch = latch;
    }

    /**
     * 异步调用完成, 回调该方法
     */
    public void onComplete(HelloService.AsyncClient.sayHello_call response) {
        try {
            System.out.println("AsyncInvokerCallback response: " + response.
getResult());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
        } finally {  
            latch.countDown();  
        }  
    }  
    /**  
     * 异步调用出错回调方法  
     */  
    public void onError(Exception exception) {  
        latch.countDown();  
    }  
}
```

pom.xml 文件 Maven 依赖内容如下：

```
<modelVersion>4.0.0</modelVersion>  
<groupId>thrift-gencode</groupId>  
<artifactId>thrift</artifactId>  
<version>1.0-SNAPSHOT</version>  
  
<properties>  
    <thrift.version>0.9.3</thrift.version>  
</properties>  
  
<dependencies>  
    <!-- thrift -->  
    <dependency>  
        <groupId>org.apache.thrift</groupId>  
        <artifactId>libfb303</artifactId>  
        <version>${thrift.version}</version>  
    </dependency>  
    <dependency>  
        <groupId>org.apache.thrift</groupId>  
        <artifactId>libthrift</artifactId>  
        <version>${thrift.version}</version>
```



```

</dependency>
<!-- junit -->
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.9</version>
</dependency>
<dependency>
<groupId>junit</groupId>
<artifactId>junit-dep</artifactId>
<version>4.9</version>
</dependency>
</dependencies>

```

整个工程的结构如图 1-10 所示。

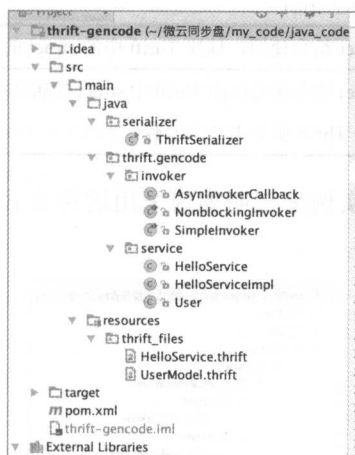


图 1-10 工程结构图

先运行 `startServer()` 方法启动服务端，然后运行 `startClient()` 发起客户端调用，完成服务的一次完整调用，运行结果如下：

```
hello, liyebingtest@163.com
```

1.4.4 基于 Java 注解的简化实现

实际操作发现，上节中自动生成的 Java 文件 `Gen-java\thrift\gencode\HelloService.java` 与 `Gen-java\thrift\gencode\User.java` 代码很复杂，可读性很差。如果我们的工程里面直接使用这样的代码，会影响整体代码的可读性与可维护性。

对于客户端与服务端都采用 Java 语言实现的 Thrift 服务，可以采用基于注解的方式简化整个开发过程，同时避免通过编写 Thrift IDL 服务描述文件来自动生成那些巨大和复杂的服务实现依赖的类文件^①。主要的 Java 注解及其作用如表 1-1 所示。

表 1-1 主要 Java 注解及其作用

注解名称	作用
<code>@ThriftService</code>	标注 Java 类为 Thrift 服务
<code>@ThriftMethod</code>	与 <code>@ThriftService</code> 结合使用，标注 Java 类中的方法为 Thrift 服务中的方法
<code>@ThriftStruct</code>	标注 Thrift 中的 struct
<code>@ThriftConstructor</code>	与 <code>@ThriftStruct</code> 结合使用，标注 Thrift 中的 struct 的构造方法
<code>@ThriftField</code>	与 <code>@ThriftStruct</code> 结合使用标注 Thrift 中 struct 的属性或者与 <code>@ThriftService</code> 及 <code>@ThriftMethod</code> 结合使用，标注 Thrift 服务中方法参数

下面通过一个具体的工程实例来说明如何采用这种方式进行开发。工程结构如图 1-11 所示。

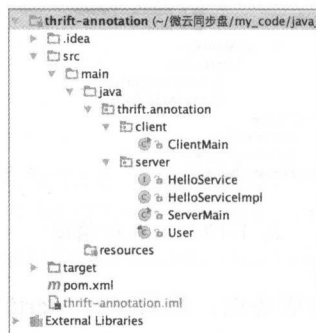


图 1-11 Thrift 注解示例工程结构

^① 该工具的 GitHub 地址：<https://github.com/facebook/swift>。

使用注解开发 Thrift 服务完整示例如下所述。

首先，通过注解标注 User.java，完成数据结构的定义，其中@ThriftStruct 标注在类上，@ThriftStruct 标注对象对应 Thrift IDL 中的 struct 类型，@ThriftField 用来标注属性，对应 Thrift IDL 中的属性定义。

```
@ThriftStruct
public final class User {

    private String name;
    private String email;

    @ThriftField(1)
    public String getName() {
        return name;
    }

    @ThriftField
    public void setName(String name) {
        this.name = name;
    }

    @ThriftField(2)
    public String getEmail() {
        return email;
    }

    @ThriftField
    public void setEmail(String email) {
        this.email = email;
    }
}
```

然后，通过@ThriftService 注解定义 Thrift 服务接口，对应 Thrift IDL 中的 service 定义。@ThriftMethod 注解标注在方法定义上，对应于 Thrift IDL service 中的方法定义。其

中方法参数需要使用@ThriftField来标注。

```
@ThriftService("HelloService")
public interface HelloService {
    @ThriftMethod
    public String sayHello(@ThriftField(name = "user") User user);
}
```

Thrift 服务接口实现如下，接口实现是一个普通的 Java Bean，不需要任何注解标注：

```
public class HelloServiceImpl implements HelloService {
    public String sayHello(User user) {
        return "hello," + user.getName() + user.getEmail();
    }
}
```

Thrift 服务启动类的代码如下，其中 ThriftServiceProcessor 对应于 Thrift Server 端的 Processor。

```
public class ServerMain {
    public static void main(String[] args) throws Exception {
        ThriftServiceProcessor processor = new ThriftServiceProcessor(
            new ThriftCodecManager(),
            new ArrayList<ThriftEventHandler>(),
            new HelloServiceImpl()
        );
        ExecutorService executorService = Executors.newFixedThreadPool(1);
        ThriftServerDef serverDef = ThriftServerDef.newBuilder()
            .listen(8899)
            .withProcessor(processor)
            .using(executorService)
            .build();

        ExecutorService bossExecutor = Executors.newCachedThreadPool();
        ExecutorService workerExecutor = Executors.newCachedThreadPool();
```

```

NettyServerConfig serverConfig = NettyServerConfig.newBuilder()
    .setBossThreadExecutor(bossExecutor)
    .setWorkerThreadExecutor(workerExecutor)
    .build();
ThriftServer server = new ThriftServer(serverConfig, serverDef);
server.start();

```

```

}

```

Thrift 服务客户端调用代码如下:

```

public class ClientMain {

    public static void main(String[] args) throws Exception {
        ThriftClientManager clientManager = new ThriftClientManager();
        FramedClientConnector connector = new FramedClientConnector(new
InetSocketAddress("localhost", 8899));

        User user = new User();
        user.setName("liyebing");
        user.setEmail("test@163.com");

        HelloService helloService = clientManager.createClient(connector
,HelloService.class).get();
        String hi= helloService.sayHello(user);
        System.out.println(hi);
    }
}

```

最后是工程的 pom.xml 配置文件:

```

<modelVersion>4.0.0</modelVersion>
<groupId>thrift-annotation</groupId>
<artifactId>thrift</artifactId>

```

```
<version>1.0-SNAPSHOT</version>
<properties>
  <swift.version>0.13.0</swift.version>
  <thrift.version>0.9.3</thrift.version>
</properties>
<dependencies>
  <!-- thrift -->
  <dependency>
    <groupId>org.apache.thrift</groupId>
    <artifactId>libfb303</artifactId>
    <version>${thrift.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.thrift</groupId>
    <artifactId>libthrift</artifactId>
    <version>${thrift.version}</version>
  </dependency>
  <!-- thrift for java annotation -->
  <dependency>
    <groupId>com.facebook.swift</groupId>
    <artifactId>swift-annotations</artifactId>
    <version>${swift.version}</version>
  </dependency>
  <dependency>
    <groupId>com.facebook.swift</groupId>
    <artifactId>swift-codec</artifactId>
    <version>${swift.version}</version>
  </dependency>
  <dependency>
    <groupId>com.facebook.swift</groupId>
    <artifactId>swift-generator</artifactId>
    <version>${swift.version}</version>
  </dependency>
```

```

<dependency>
  <groupId>com.facebook.swift</groupId>
  <artifactId>swift-idl-parser</artifactId>
  <version>${swift.version}</version>
</dependency>
<dependency>
  <groupId>com.facebook.swift</groupId>
  <artifactId>swift-service</artifactId>
  <version>${swift.version}</version>
</dependency>
<!-- logger -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.9</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.21</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
</dependencies>

```

先运行 `ServerMain.java` 启动服务，再运行类 `ClientMain.java` 调用服务。运行结果如下：

```
hello, liyebingtest@163.com
```

1.5 gRPC 介绍

gRPC 是 Google 的一个高性能、开源和通用的 RPC 框架，面向移动和 HTTP/2 设计。目前提供 C、Java 和 Go 语言版本，分别是 `grpc`、`grpc-java`、`grpc-go`。其中 C 语言版本支持 C、C++、Node.js、Python、Ruby、Objective-C、PHP 和 C#。其基于 HTTP/2 标准设计，带来诸如双向流、流控、头部压缩、单 TCP 连接上的多复用请求等特性。这些特性使得其在移动设备上表现更好，更省电和节省空间占用。序列化方式默认使用 Protocol Buffers（当然也可以使用其他数据格式，如 JSON），Protocol Buffers 是 Google 开源的一套成熟的结构数据序列化机制。

在 gRPC 里客户端应用可以像调用本地对象一样直接调用另一台不同机器上服务端应用的方法，能够更容易地创建分布式应用和服务。与许多 RPC 系统类似，gRPC 也是基于以下理念：定义一个服务，指定其能够被远程调用的方法（包含参数和返回类型）。在服务端实现这个接口，并运行一个 gRPC 服务器来处理客户端调用。在客户端拥有一个存根能够像服务端一样的方法。gRPC 客户端和服务端可以在多种环境中运行和交互——从 Google 内部的服务器到你自己的 PC，并且可以用任何 gRPC 支持的语言来编写。可以很容易地用 Java 创建一个 gRPC 服务端，用 Go、Python、Ruby 来创建客户端。此外，Google 最新 API 将有 gRPC 版本的接口，可以很容易地将 Google 的功能集成到应用里，参考 <http://doc.oschina.net/grpc?t=58008>。

gRPC 运行示意如图 1-12 所示。

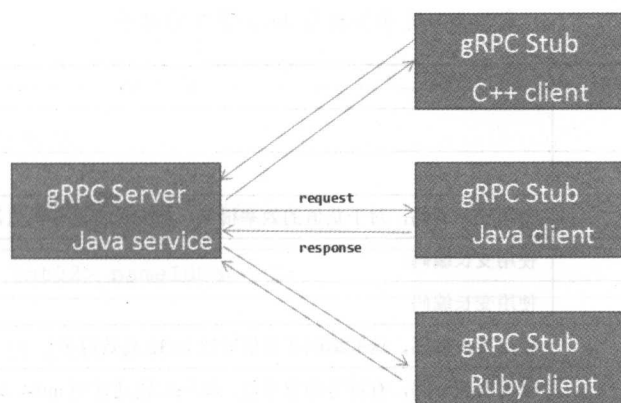


图 1-12 gRPC 运行示意图

1.5.1 protobuf3 语法介绍

protobuf3 的 IDL 主要由 message（消息）、enum（枚举）、map（映射）、service（服务）等数据结构或者元素构成。

（1）message 消息数据结构。

先看 message 的一个简单例子：

```

syntax = "proto3";

message Request {
    string name = 1;
    int32 limit = 2;
}
  
```

第 1 行 `syntax = "proto3"` 指定了正在使用 proto3 语法：如果没有指定这个，编译器会使用 proto2。这个指定语法行必须是文件的非空、非注释的第 1 行。

message 结构由一系列字段属性构成，字段类型可以是简单类型，也可以是 message 或者其他复合类型。简单类型与 Java 语言的对应关系如表 1-2 所示。

表 1-2 简单类型与 Java 语言的对应

.proto 类型	Java 类型	说 明
double	double	
float	float	
int32	int	使用变长编码，对于负值的效率很低，如果域可能有负值，请使用 sint64 替代
uint32	int	使用变长编码
uint64	long	使用变长编码
sint32	int	使用变长编码，这些编码在负值时比 int32 高效得多
sint64	long	使用变长编码，有符号的整型值，编码时比通常的 int64 高效
fixed32	int	总是 4 个字节，如果数值总是比 228 大，这个类型会比 uint32 高效
fixed64	long	总是 8 个字节，如果数值总是比 256 大，这个类型会比 uint64 高效
sfixed32	int	总是 4 个字节
sfixed64	long	总是 8 个字节
bool	boolean	
string	String	一个字符串必须是 UTF-8 编码或者 7-bit ASCII 编码的文本
bytes	ByteString	可能包含任意顺序的字节数据

完整的属性声明组成：[字段规则] 字段类型 变量名称 = 标识号。其中[字段规则]是可选的，有 **singular** 与 **repeated** 规则。其中，**singular**，一个格式良好的消息应该有 0 个或者 1 个这种字段（但是不能超过 1 个）；**repeated**，在一个格式良好的消息中，这种字段可以重复任意多次（包括 0 次），重复的值的顺序会被保留。

标识号是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改变。如果一个已有的消息格式已无法满足新的需求，需要在消息中添加一个额外的字段，同时旧版本写的代码仍然可用，这种情况下不能更改任何已有的字段的数值标识，否则会造成新旧消息格式不兼容。

（2）enum 枚举类型。

每个枚举类型必须将其第一个类型映射为 0，这个零值必须为第一个元素，为了兼容 proto2 语义，枚举类的第一个值总是默认值。例如：

```
enum STATUS{
    UNKNOWN = 0;
```

```
KNOWN= 1;
}
```

(3) map 数据类型。

map 代表一种关联映射关系。例如：

```
map<string, int32> nameIdMap= 2;
```

map 的字段可以是 repeated。

序列化后的顺序和 map 迭代器的顺序是不确定的，所以不要期望以固定顺序处理 map。当为 .proto 文件生成文本格式的时候，map 会按照 key 的顺序排序，数值化的 key 会按照数值排序。

从序列化中解析或者融合时，如果有重复的 key 则后一个 key 不会被使用。当从文本格式中解析 map 时，如果存在重复的 key，可参考“Protobuf 3 语言指南”^①。

(4) Service 服务定义。

如果想要将消息类型用在 RPC（远程方法调用）系统中，可以在 .proto 文件中定义一个 RPC 服务接口，Protocol Buffer 编译器会根据所选择的不同语言生成服务接口代码及存根。例如：

```
service HelloService {
    rpc sayHello(Request) returns (Response);
}
```

该方法能够接收 Request 并返回一个 Response。

1.5.2 gRPC 使用示例

下面我们将使用 gRPC 构建一个简单的服务发布调用示例。

^① 可参考 <http://blog.csdn.net/u011518120/article/details/54604615#Maps>。

首先添加 **Maven** 依赖及对应的 **protobuf** **Maven** 插件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>grpc</groupId>
  <artifactId>grpc-demo</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <os.detection.classifierWithLikes>debian,rhel</os.detection.classifierWithLikes>
  </properties>

  <dependencies>
    <!-- gRPC maven 依赖 -->
    <dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-netty</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-protobuf</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-stub</artifactId>
      <version>1.0.0</version>
    </dependency>
```

```

</dependencies>

<build>
<extensions>
<extension>
<groupId>kr.motd.maven</groupId>
<artifactId>os-maven-plugin</artifactId>
<version>1.4.1.Final</version>
</extension>
</extensions>
<plugins>
  <!-- protobuf-maven-plugin 插件能自动编译 protobuf IDL 文件, 生成对应 Java 代码 -->
  <plugin>
    <groupId>org.xolstice.maven.plugins</groupId>
    <artifactId>protobuf-maven-plugin</artifactId>
    <version>0.5.0</version>
    <configuration>
      <protocArtifact>com.google.protobuf:protoc:3.0.0:exe:${os.
detected.classifier}</protocArtifact>
      <pluginId>grpc-java</pluginId>
      <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.0.0:exe:${os.
detected.classifier}</pluginArtifact>
    </configuration>
  </plugin>
</plugins>
<executions>
<execution>
  <goals>
    <goal>compile</goal>
    <goal>compile-custom</goal>
  </goals>
</execution>
</executions>
</plugin>

```

```
</plugins>
</build>
</project>
```

其中，Maven 插件 `protobuf-maven-plugin` 能够通过插件运行命令自动扫描 `src/main` 目录下的 `.proto` 文件。

- ◎ `mvn protobuf: compile` 用来编译 `.proto` 文件中的 `message` 数据结构，生成对应的 Java 类。
- ◎ `mvn protobuf: compile-custom` 用来编译 `.proto` 文件中的 `service` 定义，生成对应的 `service` 基类。

其次，在 `src/main/proto` 目录下编写 `proto` 文件 `sayHello.proto`:

```
//指定 protobuf3 语法
syntax = "proto3";

option java_multiple_files = true;
option java_package = "grpc.example";
option java_outer_classname = "HelloProto";
option objc_class_prefix = "HLW";

//与 java_package 组成完整的包路径
package service;

//定义服务接口
service HelloService {
    rpc SayHello (HelloRequest) returns (HelloResponse) {}
}

//定义接口方法参数对象
message HelloRequest {
    string name = 1;
}
```

```
}

```

```
//定义接口方法返回对象

```

```
message HelloResponse {
    string message = 1;
}

```

在 pom.xml 目录下，在控制台分别运行 Maven 插件命令：

```
mvn protobuf:compile
mvn protobuf:compile-custom

```

将.proto 文件编译生成对应的 Java 类文件，如图 1-13 所示。

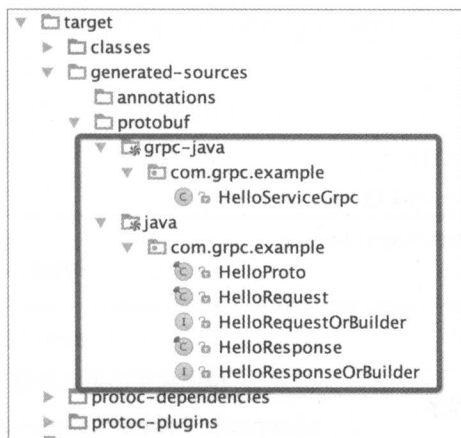


图 1-13 proto 文件编译生成对应的 Java 类文件

最后编写服务发布类及服务调用客户端，服务发布类 HelloServer：

```
public class HelloServer {
    //服务端口
    private int port = 50051;
    //服务端服务管理器
    private Server server;
    private void start() throws IOException {

```

```
//初始化并启动服务
server = ServerBuilder.forPort(port)
    .addService(new HelloServiceImpl())
    .build()
    .start();

//注册钩子, JVM 退出的时候停止服务
Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
        HelloServer.this.stop();
    }
});

//停止服务
private void stop() {
    if (server != null) {
        server.shutdown();
    }
}

// 阻塞一直到退出程序
private void blockUntilShutdown() throws InterruptedException {
    if (server != null) {
        server.awaitTermination();
    }
}

// 实现, 定义一个实现服务接口的类
private class HelloServiceImpl extends HelloServiceGrpc.
HelloServiceImplBase {
    public void sayHello(HelloRequest req, StreamObserver
```



```

<HelloResponse> responseObserver) {
    //构建返回结果对象
    HelloResponse reply = HelloResponse.newBuilder().setMessage
(("hello," + req.getName())).build();
    //将返回结果传入 stream, 返回给调用方
    responseObserver.onNext(reply);
    //通知 stream 结束
    responseObserver.onCompleted();
}
}

public static void main(String[] args) throws IOException,
InterruptedException {
    final HelloServer server = new HelloServer();
    server.start();
    server.blockUntilShutdown();
}
}

```

服务调用客户端 HelloClient:

```

public class HelloClient {
    //远程连接管理器, 管理连接的生命周期
    private final ManagedChannel channel;
    //远程服务 Stub
    private final HelloServiceGrpc.HelloServiceBlockingStub blockingStub;

    public HelloClient(String host, int port) {
        //初始化连接
        channel = ManagedChannelBuilder.forAddress(host, port)
            .usePlaintext(true)
            .build();
        //初始化远程服务 Stub
    }
}

```

```
        blockingStub = HelloServiceGrpc.newBlockingStub(channel);
    }

    public void shutdown() throws InterruptedException {
        //关闭连接
        channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);
    }

    public String sayHello(String name) {
        //构造服务调用参数对象
        HelloRequest request = HelloRequest.newBuilder().setName(name).
build();
        //调用远程服务方法
        HelloResponse response = blockingStub.sayHello(request);
        //返回值
        return response.getMessage();
    }

    public static void main(String[] args) throws InterruptedException {
        HelloClient client = new HelloClient("127.0.0.1", 50051);
        //服务调用
        String content = client.sayHello("liyebing");
        //打印调用结果
        System.out.println(content);
        //关闭连接
        client.shutdown();
    }
}
```

整个工程结构如图 1-14 所示。

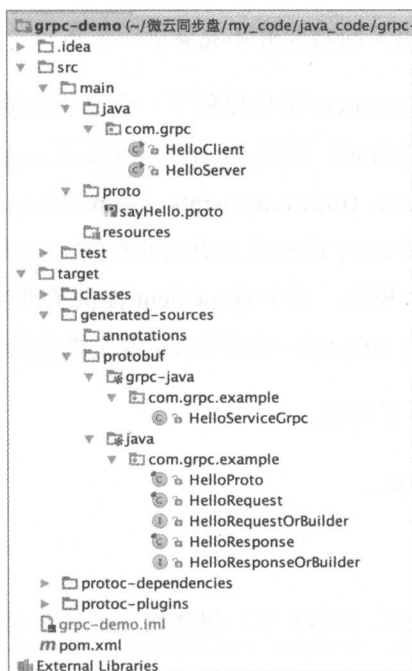


图 1-14 gRPC 示例工程结构

先运行 HelloServer 中的 main 方法，对外发布 gRPC 服务，然后运行 HelloClient 中的 main 方法，发起服务调用。运行结果如下：

```
hello, liyebing.
```

1.6 HTTP Client 介绍

超文本传输协议（HTTP）可能是现在互联网上使用的最重要的协议，可以说 HTTP 协议是整个互联网存在的基础协议之一。HTTP 协议不仅仅应用在我们通过浏览器日常上网的过程中，也广泛应用在各种系统程序之间的通信。比如，Web Services SOAP 结构数据就是使用 HTTP 协议进行传输的一种数据协议。

JDK 自带的 java.net 包里面也提供了基于 HTTP 协议最基础的实现（比如

HttpURLConnection)，但是相关的功能并不完备。

开源 Jakarta Commons HttpClient 组件提供了一个易用高效且功能丰富的 HTTP 协议实现。其中需要注意的是，HttpClient 组件 (<http://hc.apache.org/httpclient-3.x/>) 目前已经停止开发，被新的组件 Apache HttpComponents (<http://hc.apache.org/>) 所替代，Apache HttpComponents 对比旧版本的 HttpClient 具有更高的性能与灵活性。新的 HttpComponents 分为 HttpClient 与 HttpCore 两个模块。其中 HttpClient 提供了可以直接使用的面向用户方法，而 HttpCore 提供了较低层次的 HttpAPI，可以用来定制个性化的客户端与服务端 HTTP 服务。

HttpClient 的使用流程如下所述。

- (1) 构建 HttpClient 对象。
- (2) 构建 URI 对象。
- (3) 根据请求类型 (GET、POST 等) 创建相应的请求对象 (HttpGet、HttpPost 等)，并设置请求参数。
- (4) 调用 HttpClient 对象的 execute 方法发起调用。
- (5) 从返回结果中获取调用结果。

下面介绍如何使用 HttpClient。

1.6.1 构建 HttpClient 对象

HttpClient 对象有多种构建方式，既有默认值的构建方式，也可以通过设置各种参数及策略自定义 HttpClient 对象，在实际生产环境中，HttpClient 对象一般只初始化一次，作为单例对象使用，如下所示。

构建 HttpClient 对象示例 1:

```
//默认值方式构建 HttpClient 对象
CloseableHttpClient httpclient = HttpClients.createDefault();
```

下面的代码演示了如何设置常用的自定义参数初始化 HttpClient 对象。

构建 HttpClient 对象示例 2:

```
//设置常用的自定义参数初始化 HttpClient 对象
CloseableHttpClient httpClient = HttpClients.custom()
//自定义连接管理策略
.setConnectionManager(...)
//设置最大连接数
.setMaxConnTotal(...)
//自定义重试策略
.setRetryHandler(...)
//自定义拦截器
.addInterceptorFirst(...)
//自定义连接过期策略
.setKeepAliveStrategy(...).build();
```

1.6.2 构建 URI 对象

HttpClient 提供了基于 Builder 模式的 URI 构建方法,使用起来非常方便。代码示例如下:

```
//构建请求的URL, http://localhost:8080/hello/sayHello.json?userName=liyebing
URI uri = new URIBuilder()
    .setScheme("http")
    .setHost("localhost")
    .setPort(8080)
    .setPath("/hello/sayHello.json")
    .setParameter("userName", "liyebing")
    .build();
```

1.6.3 构建请求对象（HttpGet、HttpPost）

HttpClient 支持 HTTP/1.1 规范中定义的 Get、Head、Post、Put、Delete、Trace 和 Options。HttpClient 为其提供了特定的对应的对象 HttpGet、HttpHead、HttpPost、HttpPut、HttpDelete、HttpTrace 及 HttpOptions。

就实际使用而言，最常用的是 HttpGet 与 HttpPost。下面分别介绍如何构建 HttpGet 与 HttpPost：

```
//构建 HttpGet 对象
String uriGet = "http://localhost:8080/hello/sayHello.json
?userName=liyebing";
HttpGet httpGet = new HttpGet(uriGet);

//构建 HttpPost 对象
String uriPost = "http://localhost:8080/hello/sayHello.json";
HttpPost httpPost = new HttpPost(uriPost);
List<NameValuePair> nvps = new ArrayList<NameValuePair>();
nvps.add(new BasicNameValuePair("userName", "liyebing"));
httpPost.setEntity(new UrlEncodedFormEntity(nvps));
```

构建 HttpGet 对象的时候，请求参数是直接嵌入在 uri 字符串里面的，而 HttpPost 对象的请求参数则可以通过 Entity 对象设置。

1.6.4 HttpClient 发起调用及获取调用返回结果

首先介绍一个 HttpClient 调用获取调用结果的最佳实践。

自定义如下的 JSON 数据返回格式：

```
{
  "code": 0, //错误码
  "errMsg": "", //错误信息
```

```

"data": "业务数据", //业务数据
"success": true      //调用是否成功 true/false
}

```

在实际生产环境中，可以通过自定义这样一个通用 JSON 数据协议来表达 HTTP 接口所返回的数据，使得我们的代码可读性、可维护性更高。该数据格式对应的 Java 工具类如下：

```

import java.io.Serializable;

public class ApiResponse<T> implements Serializable {
    //请求是否成功 true/false
    private boolean success;
    //错误码
    private int code;
    //错误信息
    private String errMsg;
    //返回业务数据
    private T data;

    public static <T> ApiResponse<T> buildSuccess(T date) {
        ApiResponse<T> response = new ApiResponse<T>();
        response.setData(date);
        response.setSuccess(true);
        return response;
    }

    public static <T> ApiResponse<T> buildSuccess() {
        ApiResponse<T> response = new ApiResponse<T>();
        response.setSuccess(true);
        return response;
    }
}

```

```
public static <T> ApiResponse<T> buildFailure() {
    ApiResponse<T> response = new ApiResponse<T>();
    response.setSuccess(false);
    return response;
}

public static <T> ApiResponse<T> buildFailure(String errorMsg) {
    ApiResponse<T> response = new ApiResponse<T>();
    response.setSuccess(false);
    response.setCode(0);
    response.setErrMsg(errorMsg);
    return response;
}

public static <T> ApiResponse<T> buildFailure(int errorCode, String
errorMsg) {
    ApiResponse<T> response = new ApiResponse<T>();
    response.setSuccess(false);
    response.setCode(errorCode);
    response.setErrMsg(errorMsg);
    return response;
}

public boolean isSuccess() {
    return success;
}

public void setSuccess(boolean success) {
    this.success = success;
}

public int getCode() {
    return code;
}

public void setCode(int code) {
    this.code = code;
}

public String getErrMsg() {
```



```

        return errMsg;
    }

    public void setErrMsg(String errMsg) {
        this.errMsg = errMsg;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}

```

下面通过具体的代码示例演示 `HttpClient` 发起调用及获得返回结果数据。

//构建请求的 URL, 设置主机地址、端口、请求路径、请求参数等

```

URI uri = new UriBuilder().setScheme("http")
    .setHost("127.0.0.1").setPort(8080)
    .setPath("/hello/sayHello.json")
    .setParameter("userName", "liyebing").build();

```

//构建 httpGet 对象

```

HttpGet httpGet = new HttpGet(uri);

```

//请求 HTTP 接口, 并获取结果

```

CloseableHttpClient httpclient = HttpClients.createDefault();
ApiResponse response = httpclient.execute(httpGet, new ResponseHandler<ApiResponse>() {

    public ApiResponse handleResponse(HttpResponse response) throws ClientProtocolException, IOException {

        String result = EntityUtils.toString(response.getEntity(), Charset.forName("utf-8"));

        return JSON.parseObject(result, ApiResponse.class);
    }
}

```

```

    });

    //打印结果
    if (response.isSuccess()) {
        System.out.println(response.getData());
    }

```

其中，获取返回数据的时候，通过自定义 `ResponseHandler` 对象并重写 `handleResponse()` 方法来实现构建返回结果对象逻辑。通过 `FastJSON` 工具类 `JSON.parseObject()` 方法将获取到的结果字符串反序列化为 `ApiResponse` 对象。

为了提高易用性，`HttpClient` 提供了流式 API 来帮助使用者更加方便地构建 HTTP 请求。使用流式 API 来构建 HTTP 请求方法如下：

```

String uri =
    "http://localhost:8080/hello/sayHello.json?userName=liyebing";

String returnResult = Request.Get(uri)
    .execute().returnContent().asString(Charset.forName("utf-8"));

ApiResponse response = JSON.parseObject(returnResult, ApiResponse.class);
System.out.println(response.getData());

```

或者可以使用另一种写法：

```

String uri = "http://localhost:8080/hello/sayHello.json?userName=liyebing";

ApiResponse response = Request.Get(uri)
    .execute().handleResponse(new ResponseHandler<ApiResponse>() {

        public ApiResponse handleResponse(HttpResponse response) throws ClientProtocolException, IOException {
            String result = EntityUtils.toString(response.ge

```

```

tEntity(), Charset.forName("utf-8"));
        return JSON.parseObject(result, ApiResponse.class);
    }
});

System.out.println(response.getData());

```

1.7 实现自己的RPC框架

为了演示RPC的实现原理，下面我们将使用Java Socket实现一个简单的RPC框架。实现原理如图1-15所示。

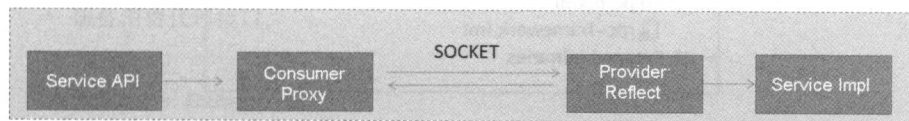


图 1-15 自定义RPC框架实现原理

整个调用流程有四个角色。

- ◎ **Service API**: 定义对外服务的接口规范。
- ◎ **Consumer Proxy**: Service API接口的代理类，内部逻辑通过Socket与服务的提供方进行通信，包括写入调用参数与获取调用返回的结果对象，通过代理使通信及获取返回结果等复杂逻辑对接口调用方透明。
- ◎ **Provider Reflect**: 服务的提供方，通过接收Consumer Proxy通过Socket写入的参数，定位到具体的服务实现，并通过Java反射技术实现服务的调用，然后将调用结果写入Socket，返回到Consumer Proxy。
- ◎ **Service Impl**: 远程服务的实现类。

下面是具体的代码实现，工程结构如图 1-16 所示。



图 1-16 自定义 RPC 框架实现工程结构

自定义 RPC 框架实现如下所述。

服务接口定义如下，对应于 Service API 角色：

```
package service;

public interface HelloService {
    public String sayHello(String content);
}
```

远程服务接口实现如下，对应于 Service Impl 角色：

```
package service;

public class HelloServiceImpl implements HelloService {
    public String sayHello(String content) {
```

```

        return "hello," + content;
    }
}

```

服务消费者代理类实现对应 Consumer Proxy 角色。通过实现服务接口的动态代理对象获得服务接口的动态代理实例 Proxy.newProxyInstance，通过实现 InvocationHandler 接口中的方法 public Object invoke (Object proxy, Method method, Object[] arguments) 来完成远程 RPC 调用。具体通过 Java 对象输出流 ObjectOutputStream 将调用接口的方法及参数写入 Socket，发起远程调用。之后通过 Java 对象输入流 ObjectInputStream 从 Socket 中获得返回结果。

```

public class ConsumerProxy {

    /**
     * 服务消费代理接口
     * @return
     * @throws Exception
     */
    public static <T> T consume(final Class<T> interfaceClass, final String host, final int port) throws Exception {
        return (T) Proxy.newProxyInstance(interfaceClass.getClassLoader(), new Class<?>[] {interfaceClass}, new InvocationHandler() {
            public Object invoke(Object proxy, Method method, Object[] arguments) throws Throwable {
                Socket socket = new Socket(host, port);
                try {
                    ObjectOutputStream output = new ObjectOutputStream(socket.getOutputStream());
                    try {
                        output.writeUTF(method.getName());
                        output.writeObject(arguments);
                        ObjectInputStream input = new ObjectInputStream(socket.getInputStream());

```

```
        try {
            Object result = input.readObject();
            if (result instanceof Throwable) {
                throw (Throwable) result;
            }
            return result;
        } finally {
            input.close();
        }
    } finally {
        output.close();
    }
} finally {
    socket.close();
}
}
});
}
```

服务发布实现对应 Provider Reflect 角色。通过 Java 对象输入流 `ObjectInputStream` 从 `Socket` 中按照 `ConsumerProxy` 的写入顺序逐一获取调用方法名称及方法参数，通过 `org.apache.commons.lang3` 中的工具方法 `MethodUtils.invokeExactMethod` 对服务实现类发起反射调用，将调用结果写入 `Socket` 返回给调用方。

```
public class ProviderReflect {
    private static final ExecutorService executorService = Executors.new
    CachedThreadPool();

    /**
     * 服务的发布
     */
    public static void provider(final Object service, int port) throws
    Exception {
```

```

ServerSocket serverSocket = new ServerSocket(port);
while (true) {
    final Socket socket = serverSocket.accept();
    executorService.execute(new Runnable() {
        public void run() {
            try {
                ObjectInputStream input=newObjectInputStream(socket.getInputStream());
                try {
                    try {
                        //方法名称
                        String methodName = input.readUTF();
                        //方法参数
                        Object[] arguments = (Object[])
input.readObject();

                        ObjectOutputStream output = new ObjectOutputStream(socket.
getOutputStream());

                        try {
                            //方法引用
                            Object result = MethodUtils.invokeExactMethod(service, methodName,
arguments);

                            output.writeObject(result);
                        } catch (Throwable t) {
                            output.writeObject(t);
                        } finally {
                            output.close();
                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                    } finally {
                        objectInputStream.close();
                    }
                } finally {

```

```
        socket.close();
    }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

});

}
```

发布服务，以便接收调用：

```
public class RpcProviderMain {
    public static void main(String[] args) throws Exception {
        HelloService service = new HelloServiceImpl();
        ProviderReflect.provider(service, 8083);
    }
}
```

服务的调用方发起调用：

```
public class RpcConsumerMain {

    public static void main(String[] args) throws Exception {
        HelloService service = ConsumerProxy.consume(HelloService.class,
"127.0.0.1", 8083);
        for (int i = 0; i < 1000; i++) {
            String hello = service.sayHello("liyebing_" + i);
            System.out.println(hello);
            Thread.sleep(1000);
        }
    }
}
```

最后 pom.xml 配置文件内容如下:

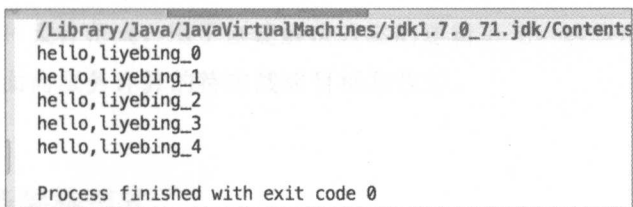
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://ma
ven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>rpc-framework</groupId>
    <artifactId>rpc</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.3.2</version>
        </dependency>
    </dependencies>

</project>
```

先运行 RpcProviderMain.java 将服务启动起来, 然后运行 RpcConsumerMain.java 进行服务的调用, 运行结果如图 1-17 所示。



```
/Library/Java/JavaVirtualMachines/jdk1.7.0_71.jdk/Contents
hello, liyebing_0
hello, liyebing_1
hello, liyebing_2
hello, liyebing_3
hello, liyebing_4
Process finished with exit code 0
```

图 1-17 运行结果

1.8 RPC 框架与分布式服务框架的区别

RPC 实现了服务消费者调用方 Client 与服务提供实现方 Server 之间的点对点调用流程，一般来说，包括 stub、通信、数据的序列化/反序列化。调用方与服务提供方一般采用直连的调用方式。

而分布式服务框架，除了包括 RPC 的特性，还包括多台 Server 提供服务的负载均衡策略及实现，服务的注册、发布与引入，以及服务的高可用策略、服务治理等特性。

1.9 本章小结

本章介绍了 RPC 框架的通用实现原理，并对平时开发过程中常用的 RPC 技术（包括 RMI、CXF、Axis2、Thrift、gRPC、HttpClient）做了介绍。最后为了进一步说明 RPC 实现原理，使用 Java Socket 实现了一个简易的 RPC 框架。RPC 框架可以看作分布式服务框架的一个功能子集，理解 RPC 框架有助于理解分布式服务框架的原理。

第2章

分布式服务框架总体架构 与功能

2.1 面向服务的体系架构（SOA）

面向服务的架构（SOA）是伴随着互联网快速发展产生的一种系统架构方法。

面向服务是一种设计范式，用于创建解决方案的逻辑单元，这些逻辑单元可组合、可复用，以支持实现面向服务计算的特定战略目标和收益。

2.1.1 面向服务架构范式

面向服务架构是面向服务解决方案的一种架构模型，具有独特特征，可支持实现面向服务原则，以及面向服务计算的战略目标。

面向服务设计范式主要由以下设计原则组成。

- ◎ 标准化服务契约：服务遵循相同的契约设计标准。
- ◎ 服务松散耦合：服务契约对服务消费者松耦合，服务之间松耦合。
- ◎ 服务抽象：服务契约仅包含必要信息，并且关于服务的信息局限为服务契约中发布的信息。
- ◎ 服务可重用性：服务可作为可重用资源。
- ◎ 服务自治：服务对其底层运行时执行环境有很大的控制权。
- ◎ 服务无状态性：服务无状态保证了服务部署的横向扩展性。
- ◎ 服务可发现性：服务可以通过描述性元数据有效发现并解释服务。
- ◎ 服务可组合性：可以通过组合叠加原子服务形成复杂上层业务服务。

SOA 中的服务化收益^①如图 2-1 所示。

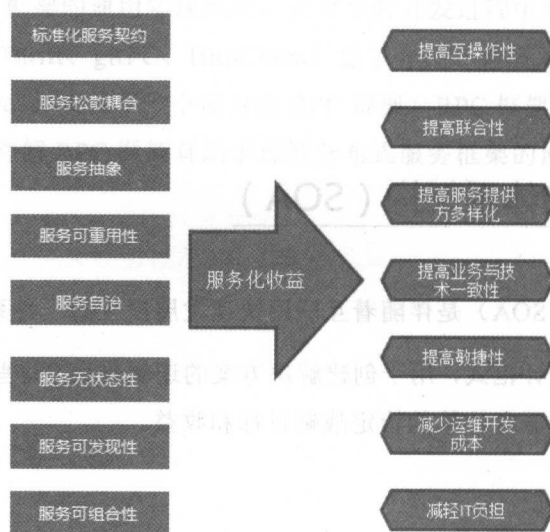


图 2-1 SOA 中的服务化收益

^① 参考 *SOA with Java: Realizing Service-Oriented Architecture with Java Technologies*。

2.1.2 服务拆分原则

在互联网企业，随着业务的不断扩张，快速演进，单一的系统已经不足以承载大量的业务。如果把所有的业务实现在一个单一的系统应用中会产生以下问题。

- ◎ 业务模块边界不清，代码耦合严重，无法很好地实现代码模块级别或者功能级别的复用，进而无法快速地进行业务迭代。
- ◎ 所有的开发人员都在一个应用工程代码库进行迭代开发、测试、发版，会导致应用发布上线过于频繁，不利于线上系统稳定性，而且整个应用代码的稳定性、可维护性都很难得到保障。
- ◎ 因为不同业务实现之间没有拆分离部署，某些高 QPS 耗时较长的复杂操作会影响整体应用的可用性、系统伸缩性。

解决以上问题的一个很自然的做法就是做系统拆分，以服务化的思想为指导，实现面向服务的架构，做服务化拆分。

服务化首先要解决的是如何去拆解我们的系统。一般首先需要从整体上梳理公司的业务，包括过去、现在及未来可以预见几年内的业务进展，梳理公司的发展战略。将业务模块化，分解出各个业务模块之间的依赖及业务模块之间的边界。按照业务边界及业务之间的依赖顺序进行系统拆分，这是系统拆分的常见做法。通过系统拆分实现 SOA 架构的价值，沉淀一批稳定的后端服务，通过叠加复用快速响应用户的前端需求。

从单体应用到 SOA 架构的转变如图 2-2 所示。

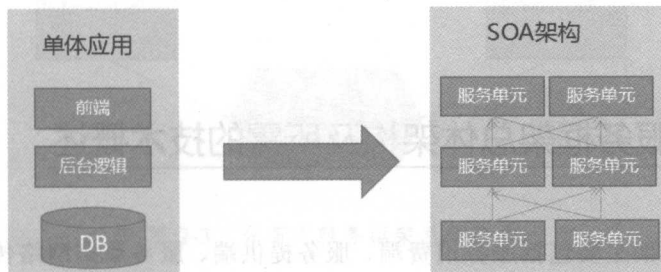


图 2-2 从单体应用到 SOA 架构的转变

面向服务的架构的交付价值，一方面通过叠加复用已有服务，实现业务敏捷快速地响应业务需求。另一方面让系统边界与业务边界对齐，使得系统架构不仅能够满足当前业务需要，而且能够面向未来与公司战略对齐，使得系统能够持续稳定地输出对业务的支持。撇开业务，从系统实现本身来说，服务化能够将许多复杂系统的伸缩性简化为某个或某几个服务组的伸缩性，使得我们能够通过专注于解决单个服务的伸缩性进而解决复杂系统整体的伸缩性。

2.2 分布式服务框架现实需求

SOA 面向服务的架构体系所需要解决的技术问题中最重要也是最基本的，便是如何实现服务之间的通信。第 1 章介绍了多种系统之间 RPC 调用框架的使用，如 RMI、CXF、Axis2、Thrift、gRPC、HttpClient 等。但是此类 RPC 框架并没有实现服务治理机制。一般来说，服务治理包括服务依赖梳理、负载均衡、服务分组灰度发布、链路监控、服务质量统计、服务自动发现、自动下线与服务注册中心等功能。

可以简单地认为 RPC 框架与服务治理组合起来，就构成了一套完整的分布式服务框架。实现 SOA 架构最重要的便是选择一套合适的分布式服务框架，分布式服务框架是 SOA 能够最终成功落地实现价值交付的技术保障。

RPC 框架实现 SOA 服务化之后服务之间的通信，服务治理实现服务运维，指导服务化更好地快速持续演进。没有服务治理，服务化进程好比蒙眼狂奔的重卡，随时会有翻车的危险。

2.3 分布式服务框架总体架构及所需的技术概述

分布式服务框架主要包括服务消费端、服务提供端、服务数据网络传输的序列化与反序列化、服务数据的通信机制、服务注册中心、服务治理这几个组成部分。

Spring 作为 Java 企业级开发事实上的开发标准，一般使用 Spring 来实现服务的引入

与发布，分别实现服务消费端与服务提供端。

服务数据的序列化与反序列化实现有多种选择，比如 Java 默认序列化、XML 格式、JSON 格式、Hessian、protostuff、Thrift、Avro 等，可以根据我们对性能的要求及实际的使用场景来选择合适的序列化方式。

出于性能（健壮性与易用性）考虑，服务数据的通信机制目前的主流实现一般采用 Mina、Grizzly、Netty 或者类似的 NIO 网络通信框架。

服务注册中心目前的主流是采用 ZooKeeper 来实现，用来实现服务注册、服务发现、服务自动上下线等功能。后面会分为多个章节分别对以上技术进行详细解读。

如图 2-3 所示，分布式服务框架由服务提供端、服务消费端、服务注册中心、服务治理四部分组成。

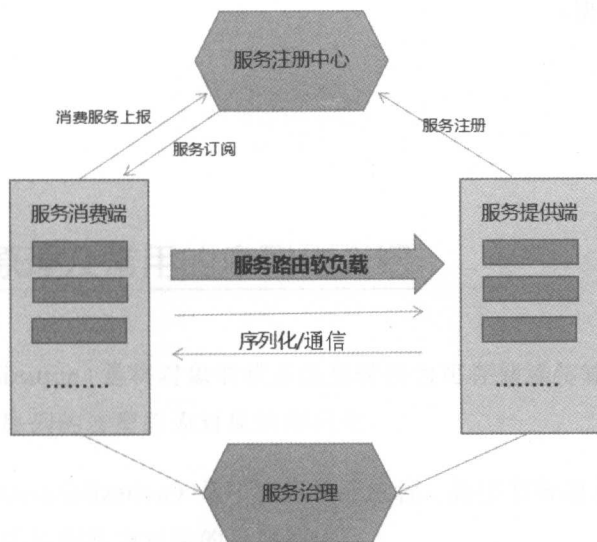


图 2-3 分布式服务框架总体架构

- ◎ 服务提供端启动服务，将服务提供者信息注册到服务注册中心，服务提供者信息一般包括服务主机地址、端口、服务接口信息等。

- ◎ 服务消费端将服务提供者信息从服务注册中心获取到本地缓存，同时将服务消费者信息上报到服务注册中心。
- ◎ 服务消费端根据某种软负载策略选择某一个服务提供者发起服务调用，服务调用首先采用某种数据序列化方案，将调用数据序列化为可以在网络传输的字节数组，采用某种 NIO 框架（如 Netty、Mina、Grizzly）完成调用。
- ◎ 为了管理大规模的服务依赖关系，需要提供服务治理功能。

2.4 本章小结

本章从整体上概述了面向服务的架构体系及分布式服务框架的总体架构，是后面实现分布式服务框架的依据。

第3章

分布式服务框架序列化与反序列化实现

3.1 序列化原理及常用的序列化介绍

序列化（Serialization）是将对象的状态信息转换为可存储或传输的形式过程。简言之，把对象转换为字节序列的过程称为对象的序列化。

而反序列化（Deserialization）是序列化的逆过程。将字节数组反序列化为对象，把字节序列恢复为对象的过程称为对象的反序列化。

序列化能帮助我们解决如下问题。

- ◎ 通过将对象序列化为字节数组，使得不共享内存通过网络连接的系统之间能够进行对象的传输。

◎ 通过将对象序列化为字节数组，能够将对象永久存储到存储设备。

◎ 解决远程接口调用 JVM 之间内存无法共享的问题。

评价一个序列化算法优劣的两个重要指标。

◎ 序列化后码流的大小。

◎ 序列化本身的速度及系统资源开销大小（包括内存、CPU 等）。

各种常用的序列化的性能对比^①如图 3-1 所示。

pre.	create	ser	deser	total	size	+df1
protostuff	112	663	917	1580	239	150
fst-flat-pre	81	908	984	1893	251	165
kryo-flat-pre	81	831	1083	1914	212	132
protobuf	149	1625	942	2567	239	149
msgpack-databind	81	1123	1821	2944	233	146
thrift-compact	139	1844	1101	2945	240	148
json/fastjson/databind	82	1710	1633	3343	486	262
thrift	139	2216	1253	3469	349	197
scala/sbinary	166	2122	1451	3573	255	147
smile/jackson+afterburner/databind	81	1757	1868	3625	352	252
smile/jackson/databind	81	2095	2250	4346	338	241
json/jackson+afterburner/databind	81	2006	2548	4554	485	261
json/protostuff-runtime	80	2008	2741	4748	469	243
json/jackson/databind	80	2173	2972	5145	485	261
json/jackson-jr/databind	81	2282	3435	5716	468	255
cbor/jackson/databind	81	4112	2659	6771	397	246
xml/jackson/databind	81	3558	7028	10585	683	286
json/gson/databind	81	7322	7063	14386	486	259
bson/jackson/databind	80	6974	8318	15291	506	286
xml/xstream+c	81	9050	28265	37315	487	244
xml/exi-manual	83	19634	18063	37697	337	327
json/javax-tree/glassfish	1558	16804	23814	40618	485	263
java-built-in	82	7154	37804	44958	889	514
scala/java-built-in	164	11195	62423	73617	1312	700
json/protobuf	142	11815	73133	84949	488	253
json/json-lib/databind	81	45857	165134	210991	485	263

图 3-1 常用序列化性能对比

① 参考 <https://github.com/eishay/jvm-serializers/wiki>。

为了抽象出一个序列化/反序列化通用服务，首先定义序列化/反序列化通用接口，代码如下：

```
public interface ISerializer {  
    /**  
     * 序列化  
     *  
     * @param obj  
     * @param <T>  
     * @return  
     */  
    public <T> byte[] serialize(T obj);  
  
    /**  
     * 反序列化  
     *  
     * @param data  
     * @param clazz  
     * @param <T>  
     * @return  
     */  
    public <T> T deserialize(byte[] data, Class<T> clazz);  
}
```

下面将基于该接口，给出日常开发中常见的序列化/反序列化实现。

3.2 Java 默认的序列化

JDK 提供了 Java 对象的序列化方式。Java 的序列化主要通过对象输出流 `java.io.ObjectOutputStream` 与对象输入流 `java.io.ObjectInputStream` 来实现，其中被序列化的类需要实现 `java.io.Serializable` 接口，如下所示。

```

public class DefaultJavaSerializer implements ISerializer {

    @SuppressWarnings("unchecked")
    public <T> byte[] serialize(T obj) {
        ByteArrayOutputStream byteArrayOutputStream = new
        ByteArrayOutputStream();
        try {
            ObjectOutputStream objectOutputStream = new ObjectOutputStream
            (byteArrayOutputStream);
            objectOutputStream.writeObject(obj);

            objectOutputStream.close();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return byteArrayOutputStream.toByteArray();
    }

    @SuppressWarnings("unchecked")
    public <T> T deserialize(byte[] data, Class<T> clazz) {

        ByteArrayInputStream byteArrayInputStream = new
        ByteArrayInputStream(data);
        try {
            ObjectInputStream objectInputStream = new
            ObjectInputStream(byteArrayInputStream);
            return (T) objectInputStream.readObject();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

以上的代码给出了使用 Java 进行序列化与反序列化的通用代码。

关于 Java 序列化还有如下一些知识点。

- ◎ 序列化时，只对对象的状态进行保存，而不管对象的方法。
- ◎ 当一个父类实现序列化，子类自动实现序列化，不需要显式实现 `Serializable` 接口。
- ◎ 当一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化。
- ◎ 当某个字段被声明为 `transient` 后，默认序列化机制就会忽略该字段。
- ◎ 对于上述已被声明为 `transient` 的字段，可以在类中添加私有方法 `writeObject()` 与 `readObject()` 两个方法来进行序列化。

绕过 `transient` 机制实现序列化/反序列化：

```
public class User implements Serializable {
    private transient Integer age = 28;
    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        out.writeInt(age);
    }

    private void readObject(ObjectInputStream in) throws IOException,
        ClassNotFoundException {
        in.defaultReadObject();
        age = in.readInt();
    }
}
```

上述代码使得 `age` 属性值虽然被关键字 `transient` 修饰，但依然能够正确地被序列化与反序列化。

无论是使用 `transient` 关键字，还是使用 `writeObject()` 和 `readObject()` 方法，其实都是基

于 `Serializable` 接口的序列化。JDK 中提供了另一个序列化接口 `Externalizable`, `Externalizable` 继承于 `Serializable`, 当使用该接口时, 序列化的细节需要由程序员自主完成, 适用于有特殊定制化需求的场景。

Java 默认的序列化机制优缺点都非常明显。

优点:

- ◎ Java 语言自带, 无须额外引入第三方依赖。
- ◎ 与 Java 语言有天然的最好的易用性与亲和性。

缺点:

- ◎ 只支持 Java 语言, 不支持跨语言。
- ◎ Java 默认序列化性能欠佳, 序列化后产生的码流过大, 对于引用过深的对象序列化易发生内存 OOM 异常。

3.3 XML 序列化框架介绍

XML 序列化的优势在于可读性好, 利于调试。因为使用标签对来表示数据, 导致序列化后码流大, 而且效率不高。适用于对性能要求不高, 且 QPS 较低的企业级内部系统之间的数据交换的场景, 又因 XML 具有语言无关性, 可用于异构系统之间的数据交换协议。其中我们熟悉的 `WebService` 相关的协议就是采用 XML 格式对数据进行序列化的。

XML 序列化/反序列化有多种实现方式。这里介绍 `XStream` 与 Java 自带的 XML 序列化/反序列化两种方式。

首先介绍 `XStream` 实现 XML 序列化/反序列化:

```
public class XmlSerializer implements ISerializer {  
    //初始化 XStream 对象
```

```

private static final XStream xStream = new XStream(new DomDriver());

@SuppressWarnings("unchecked")
public <T> byte[] serialize(T obj) {
    return xStream.toXML(obj).getBytes();
}

@SuppressWarnings("unchecked")
public <T> T deserialize(byte[] data, Class<T> clazz) {
    String xml = new String(data);
    return (T) xStream.fromXML(xml);
}
}

```

其中，XML 序列化反序列化使用了 XStream 开源工具包。关于 XStream 在此不做介绍，请自行查看官网介绍 <http://x-stream.github.io/>。

Maven 配置如下：

```

<dependency>
<groupId>com.thoughtworks.xstream</groupId>
<artifactId>xstream</artifactId>
<version>1.4.9</version>
</dependency>

```

Java 自带方式实现 XML 序列化/反序列化主要是使用 `java.beans.XMLEncoder` 与 `java.beans.XMLDecoder` 类完成相应的功能。

```

public class XML2Serializer implements ISerializer {
    @Override
    public <T> byte[] serialize(T obj) {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        XMLEncoder xe = new XMLEncoder(out, "utf-8", true, 0);
        xe.writeObject(obj);
    }
}

```

```
        xe.close();
        return out.toByteArray();
    }

    @Override
    public <T> T deserialize(byte[] data, Class<T> clazz) {
        XMLDecoder xd = new XMLDecoder(new ByteArrayInputStream(data));
        Object obj = xd.readObject();
        xd.close();
        return (T) obj;
    }
}
```

3.4 JSON 序列化框架介绍

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。相比 XML, JSON 码流更小, 而且保留了 XML 可读性好的优势。

JSON 序列化常用的开源工具有如下几个。

- ◎ Jackson (官网 <https://github.com/FasterXML/jackson/>)。
- ◎ 阿里开源的 fastjson (官网 <https://github.com/alibaba/fastjson>)。
- ◎ Google 开发的 GSON (官网 <https://github.com/google/gson>)。

相比较而言, Jackson 与 fastjson 比 GSON 的性能要好。Jackson、GSON 相对 fastjson 稳定性更好, fastjson 对序列化对象本身有一些额外的要求, 比如序列化对象的属性必须实现 get/set 方法才能完成对该属性的序列化。fastjson 的优势在于非常易用的 API 操作及高性能。

下面使用 Jackson 来给出 JSON 序列化的通用实现方法, 关键代码如下:


```
public class JsonSerializer implements ISerializer {

    private static final ObjectMapper objectMapper = new ObjectMapper();
    static {
        objectMapper.configure(JsonParser.Feature.ALLOW_COMMENTS, true);
        objectMapper.configure(JsonParser.Feature.ALLOW_UNQUOTED_FIELD_
NAMES, true);
        objectMapper.configure(JsonParser.Feature.ALLOW_SINGLE_QUOTES,
true);
        objectMapper.configure(JsonParser.Feature.ALLOW_UNQUOTED_
CONTROL_CHARS, true);
        objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_
PROPERTIES, false);
        objectMapper.configure(SerializationFeature.FAIL_ON_EMPTY_BEANS,
false);

        SimpleModule module = new SimpleModule("DateTimeModule",
Version.unknownVersion());
        module.addSerializer(Date.class, new FDateJsonSerializer());
        module.addDeserializer(Date.class, new FDateJsonDeserializer());
        objectMapper.registerModule(module);
    }
    private static ObjectMapper getObjectMapperInstance() {
        return objectMapper;
    }

    @SuppressWarnings("unchecked")
    public <T> byte[] serialize(T obj) {
        if (obj == null) {
            return new byte[0];
        }
        try {
            String json = objectMapper.writeValueAsString(obj);
            return json.getBytes();
        }
    }
}
```

```
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
  
    @SuppressWarnings("unchecked")  
    public <T> T deserialize(byte[] data, Class<T> clazz) {  
        String json = new String(data);  
        try {  
            return (T) objectMapper.readValue(json, clazz);  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

其中 `FDateJsonSerializer.java` 与 `FDateJsonDeserializer.java` 用来对 `java.util.Date` 类进行序列化格式的定制化输出，代码如下：

```
public class FDateJsonDeserializer extends JsonSerializer<Date> {  
    @Override  
    public Date deserialize(JsonParser gen, DeserializationContext ctxt)  
        throws IOException, JsonProcessingException {  
        String date = gen.getText();  
        if (StringUtils.isEmpty(date)) {  
            return null;  
        }  
        if (StringUtils.isNumeric(date)) {  
            return new Date(Long.valueOf(date));  
        }  
        try {  
            SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd  
HH:mm:ss");  
            return format.parse(date);  
        }  
    }  
}
```

```

        } catch (Exception e) {
            throw new IOException(e);
        }
    }
}

public class FDateJsonSerializer extends JsonSerializer<Date> {
    @Override
    public void serialize(Date date, JsonGenerator jsonGenerator,
        SerializerProvider serializers) throws IOException, JsonProcessingException {
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        jsonGenerator.writeString(date != null ? format.format(date) :
        "null");
    }
}

```

Jackson Maven 依赖配置如下:

```

<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-core</artifactId>
<version>2.8.6</version>
</dependency>
<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
<version>2.8.6</version>
</dependency>
<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-annotations</artifactId>
<version>2.8.6</version>
</dependency>

```

如果采用 `fastjson` 来实现序列化/反序列化代码就简单多了。原因在于 `fastjson` 提供了非常简洁好用的 API 及工具类。下面是采用 `fastjson` 实现序列化/反序列化的代码示例：

```
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.serializer.SerializerFeature;
import serializer.ISerializer;

public class JSON2Serializer implements ISerializer {

    @Override
    public <T> byte[] serialize(T obj) {
        JSON.DEFFAULT_DATE_FORMAT = "yyyy-MM-dd HH:mm:ss";
        return JSON.toJSONString(obj, SerializerFeature.WriteDateUseDateFormat).getBytes();
    }

    @Override
    public <T> T deserialize(byte[] data, Class<T> clazz) {
        return (T) JSON.parseObject(new String(data), clazz);
    }
}
```

`fastjson` Maven 依赖配置如下：

```
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>fastjson</artifactId>
<version>1.2.29</version>
</dependency>
```

3.5 Hessian 序列化框架介绍

Hessian (官网 <http://hessian.caucho.com/>) 是一个支持跨语言传输的二进制序列化协议。相对于 Java 默认的序列化机制, Hessian 具有更好的性能与易用性, 而且支持多种不同的语言。

其中, AbstractSerializerFactory、AbstractHessianOutput、AbstractSerializer、AbstractHessianInput、AbstractDeserializer 是 Hessian 实现序列化和反序列化的核心类, 示例如下:

```
public class HessianSerializer implements ISerializer {

    public byte[] serialize(Object obj) {
        if (obj == null)
            throw new NullPointerException();

        try {
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            HessianOutput ho = new HessianOutput(os);
            ho.writeObject(obj);
            return os.toByteArray();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @SuppressWarnings("unchecked")
    public <T> T deserialize(byte[] data, Class<T> clazz) {
        if (data == null)
            throw new NullPointerException();
    }
}
```

```
try {  
    ByteArrayInputStream is = new ByteArrayInputStream(data);  
    HessianInput hi = new HessianInput(is);  
    return (T) hi.readObject();  
} catch (Exception e) {  
    throw new RuntimeException(e);  
}  
}
```

Hessian Maven 依赖配置如下：

```
<dependency>  
<groupId>com.caucho</groupId>  
<artifactId>hessian</artifactId>  
<version>4.0.38</version>  
</dependency>
```

3.6 protobuf 序列化框架介绍

protobuf 是 Google 的一种数据交换的格式，它独立于语言，独立于平台。Google 提供了多种语言的实现：Java、C#、C++、Go 和 Python，每一种实现都包含了相应语言的编译器及库文件。protobuf 是一个纯粹的展示层协议，可以和各种传输层协议一起使用。protobuf 的文档也非常完善。

protobuf 具有广泛的用户基础，空间开销小及高解析性能是其亮点，非常适合于公司内部对性能要求高的 RPC 调用。由于其解析性能高，序列化后数据量相对少，也适合应用层对象的持久化场景。

它的主要问题在于需要编写 .proto IDL 文件，使用起来工作量稍大，且需要额外学习

proto IDL 特有的语法,增加了额外的学习成本。

使用 protobuf 的一般步骤如下。

(1) 配置开发环境,安装 Protocol Compiler 代码编译器。

(2) 编写 .proto 文件,定义序列化对象的数据结构。

(3) 基于编写的 .proto 文件,使用 Protocol Compiler 编译生成对应的序列化/反序列化工具类。

(4) 基于自动生成的代码,编写自己的序列化应用。

下面将按照上述 4 个步骤,以 Java 语言为例提供一个完整的例子,演示如何使用 protobuf 协议进行数据序列化/反序列操作。

第 1 步:以 Mac 系统为例,在 <https://github.com/google/protobuf/releases/tag/v3.0.0> 下载 protoc-3.0.0-osx-x86_64.zip,解压到当前目录,进入目录 protoc-3.0.0-osx-x86_64/bin,发现有可执行文件 protoc,可以使用 protoc 来自动生成我们所需的序列化基础工具类。

第 2 步:编写文件 addressbook.proto 如下(此处使用 proto2 版本的语法)。

```
syntax = "proto2";  
package model;  
  
option java_package = "model";  
option java_outer_classname = "AddressBookProtos";  
  
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
    enum PhoneType {  
        MOBILE = 0;  
        HOME = 1;
```

```
WORK = 2;
}

message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
}

repeated PhoneNumber phone = 4;
}

message AddressBook {
    repeated Person person = 1;
}
```

第 3 步：执行如下命令，生成相关的序列化工具类。

```
./protoc --java_out=./ ./addressbook.proto
```

执行成功后，可以看到会在当前目录下生成文件 `model/AddressBookProtos.java`。

第 4 步：将生成的代码类 `AddressBookProtos.java` 复制到工程里面，编写应用的序列化与反序列化工具类。

```
public static void main(String[] args) throws Exception {
    //构建一个 Person 对象
    AddressBookProtos.Person person = AddressBookProtos.Person
        .newBuilder()
        .setEmail("kongxuan@163.com")
        .setId(10000)
        .setName("kongxuan")
        .addPhone(
            AddressBookProtos.Person.PhoneNumber.newBuilder()
                .setNumber("13300000000")
                .setType(AddressBookProtos.Person.PhoneType.HOME)
            )
    }
```



```

        .build()).build();

//序列化方法一
System.out.println(person.toByteString());

//序列化方法二
System.out.println(Arrays.toString(person.toByteArray()));

//反序列化方法一
AddressBookProtos.Person newPerson =
AddressBookProtos.Person.parseFrom(person.toByteString());
    System.out.println(newPerson);

//反序列化方法二
newPerson = AddressBookProtos.Person.parseFrom(person.toByteArray());
System.out.println(newPerson);
}

```

其中 `person.toByteString()` 将 `person` 对象序列化为 `ByteString` 类型的对象，`person.toByteArray()` 将 `person` 对象序列化为 `byte[]`。与之对应，反序列化方法 `AddressBookProtos.Person.parseFrom(person.toByteString())` 将 `ByteString` 类型对象反序列化为 `AddressBookProtos.Person` 对象，`AddressBookProtos.Person.parseFrom(person.toByteArray())` 将 `byte[]` 反序列化为 `AddressBookProtos.Person` 对象。

在以上代码的基础上，我们也可以利用 Java 反射抽取一个 `protobuf` 对象序列化/反序列化的通用方法。代码如下：

```

import com.google.protobuf.GeneratedMessageV3;
import model.AddressBookProtos;
import org.apache.commons.lang3.reflect.MethodUtils;
import serializer.ISerializer;
import java.util.Arrays;

public class ProtoBufSerializer implements ISerializer {
    public <T> byte[] serialize(T obj) {
        try {

```

```
        if (!(obj instanceof GeneratedMessageV3)) {
            throw new UnsupportedOperationException("not supported obj
type");
        }
        return (byte[]) MethodUtils.invokeMethod(obj, "toByteArray");
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

public <T> T deserialize(byte[] data, Class<T> cls) {
    try {
        if (!GeneratedMessageV3.class.isAssignableFrom(cls)) {
            throw new UnsupportedOperationException("not supported obj
type");
        }

        Object o = MethodUtils.invokeStaticMethod(cls,
"getDefaultInstance");
        return (T) MethodUtils.invokeMethod(o, "parseFrom", new
Object[]{data});
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
}
```

下面是该序列化/反序列化通用方法使用示例：

```
public static void main(String[] args) throws Exception {
    //构建一个 Person 对象
    AddressBookProtos.Person person = AddressBookProtos.Person.newBuilder()
```

```

.setEmail("kongxuan@163.com")
.setId(10000)
.setName("kongxuan")
.addPhone(AddressBookProtos.Person.PhoneNumber.newBuilder().setNumber("1
3300000000").setType(AddressBookProtos.Person.PhoneType.HOME).build()).build
());
ProtoBufSerializer serializer =new ProtoBufSerializer ();
//序列化
byte[] data = serializer.serialize(person);
//反序列化
AddressBookProtos.Person personCopy =      serializer.deserialize(data,
AddressBookProtos.Person.class);
System.out.println(personCopy);
}

```

其中 protobuf 及 apache.commons 的 Maven 依赖配置如下:

```

<dependency>
<groupId>com.google.protobuf</groupId>
<artifactId>protobuf-java</artifactId>
<version>3.1.0</version>
</dependency>
<dependency>
<groupId>org.apache.commons</groupId>
<artifactId>commons-lang3</artifactId>
<version>3.3.2</version>
</dependency>

```

3.7 protostuff 序列化框架介绍

通过上一节的学习,我们知道 protobuf 需要预先编写 proto IDL 文件,再通过 protobuf

提供的编译器生成对应于各种语言的代码。但是对于 Java 语言来说，Java 具有反射和动态代码生成的能力，这个预编译过程不是必需的，可以在代码执行时实现。而 `protostuff` 就实现了该功能。`protostuff` 基于 Google `protobuf`，其中，`protostuff-runtime` 实现了无须预编译对 Java Bean 进行 `protobuf` 序列化/反序列化的能力。

对于仅使用 Java 语言，且无须跨语言的使用场景，`protostuff` 继承了 Google `protobuf` 的高性能的同时免去了编写 `.proto` 文件的麻烦，是非常值得推荐的序列化/反序列化方案。

下面给出通用的基于 `protostuff` 实现的序列化/反序列化工具方法。

```
public class ProtoStuffSerializer implements ISerializer {

    private static Map<Class<?>, Schema<?>> cachedSchema = new
    ConcurrentHashMap<Class<?>, Schema<?>>();

    private static Objenesis objenesis = new ObjenesisStd(true);

    @SuppressWarnings("unchecked")
    private static <T> Schema<T> getSchema(Class<T> cls) {
        Schema<T> schema = (Schema<T>) cachedSchema.get(cls);
        if (schema == null) {
            schema = RuntimeSchema.createFrom(cls);
            cachedSchema.put(cls, schema);
        }
        return schema;
    }

    @SuppressWarnings("unchecked")
    public <T> byte[] serialize(T obj) {
        Class<T> cls = (Class<T>) obj.getClass();
        LinkerBuffer buffer = LinkerBuffer.allocate(LinkerBuffer.DEFAULT_
        BUFFER_SIZE);
        try {
            Schema<T> schema = getSchema(cls);
```

```

        return ProtostuffIOUtil.toByteArray(obj, schema, buffer);
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        buffer.clear();
    }
}

public <T> T deserialize(byte[] data, Class<T> cls) {
    try {
        T message = (T) objenesis.newInstance(cls);
        Schema<T> schema = getSchema(cls);
        ProtostuffIOUtil.mergeFrom(data, message, schema);
        return message;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

以上代码对应的 Maven 依赖配置如下:

```

<dependency>
<groupId>com.dyuproject.protostuff</groupId>
<artifactId>protostuff-core</artifactId>
<version>1.0.9</version>
</dependency>
<dependency>
<groupId>com.dyuproject.protostuff</groupId>
<artifactId>protostuff-runtime</artifactId>
<version>1.0.9</version>
</dependency>
<dependency>

```

```
<groupId>org.objenesis</groupId>
<artifactId>objenesis</artifactId>
<version>2.5.1</version>
</dependency>
```

需要说明的是，以上代码在反序列化实例化对象的时候，使用了 **Objenesis** 来完成对象的实例化。

```
T message = (T) objenesis.newInstance(cls);
```

如果是使用 **Java** 反射来实例化对象，代码如下：

```
T message = (T) cls.getConstructors()[0].newInstance();
```

这种做法要求 **Java** 对象必须保留无参构造函数，且调用构造函数不会造成额外的副作用（如构造函数内部实现特殊业务逻辑、抛出异常等），由此代码的通用性和健壮性受到了影响。

使用 **Objenesis** 可以绕过构造器来创建实例。根据 **JVM** 供应商、版本和类的安全管理和类型的不同，**Objenesis** 实例化对象有许多不同的策略。

◎ **Standard**：没有构造器会被调用。

◎ **Serializable compliant**：与 **Java** 标准序列化方式实例化一个对象的行为一致。

最简单的使用 **Objenesis** 的方法是使用 **ObjenesisStd**（**Standard**）和 **ObjenesisSerializer**（**Serializable compliant**）。这种方式会自动选择最好的策略，代码如下。

```
import org.objenesis.Objenesis;
import org.objenesis.ObjenesisStd;
import org.objenesis.instantiator.ObjectInstantiator;

public class ObjenesisDemo {
    public static void main(String[] args) {
        //实例化对象第1种方式
        Objenesis objenesis1 = new ObjenesisStd();
```

```

        MessageInfo messageInfo1 = (MessageInfo) objenesis1.newInstance
(MessageInfo.class);
        System.out.println(messageInfo1.getClass());

        //实例化对象第2种方式 (ObjectInstantiator 线程安全, 可复用, 可提高性能)
        Objenesis objenesis2 = new ObjenesisStd();
        ObjectInstantiator instantiator =
objenesis2.getInstantiatorOf(MessageInfo.class);
        MessageInfo messageInfo2 =
instantiator.newInstance();
        System.out.println(messageInfo2.getClass());
    }

    class MessageInfo {
        //去除默认的无参构造函数
        public MessageInfo(long messageId) {
            this.messageId = messageId;
        }
        private long messageId;
        public long getMessageId() {
            return messageId;
        }
        public void setMessageId(long messageId) {
            this.messageId = messageId;
        }
    }
}

```

其中类 `MessageInfo` 只保留了有参构造函数，去除了无参默认构造函数。

运行结果如下：

```

class ObjenesisDemo$MessageInfo
class ObjenesisDemo$MessageInfo

```

3.8 Thrift 序列化框架介绍

在第 1 章里面已经介绍了 Thrift RPC 框架，实际上也可以单独使用 Thrift 进行数据序列化/反序列化操作。

与 protobuf 类似，使用 Thrift 之前，需要编写以 .thrift 结尾的 IDL 文件，再使用 Thrift 提供的编译器编译生成对应的代码。对 Java 而言，所有生成的 Java Bean 都继承了类 `org.apache.thrift.TBase`。

使用 Thrift 的序列化机制，步骤如下。

(1) 编写所需要序列化数据结构的 .thrift 文件。

(2) 使用 Thrift 提供的代码生成工具，生成 .thrift 文件对应的 Java 类。

(3) 使用 Thrift 提供的 `org.apache.thrift.TSerializer` 与 `org.apache.thrift.TDeserializer` 分别对类对象进行序列化与反序列化操作，代码如下。

```
public class ThriftSerializer implements ISerializer {

    public <T> byte[] serialize(T obj) {
        try {
            if (!(obj instanceof TBase)) {
                throw new UnsupportedOperationException("not supported obj
type");
            }

            TSerializer serializer = new TSerializer(new TBinaryProtocol.
Factory());

            return serializer.serialize((TBase) obj);
        } catch (TException e) {
            throw new RuntimeException(e);
        }
    }
}
```



```

    }
}

public <T> T deserialize(byte[] data, Class<T> cls) {
    try {
        if (!TBase.class.isAssignableFrom(cls)) {
            throw new UnsupportedOperationException("not supported obj
type");
        }
        TBase o = (TBase) cls.newInstance();
        TDeserializer tDeserializer = new TDeserializer(new
TBinaryProtocol.Factory());
        tDeserializer.deserialize(o, data);
        return (T) o;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

其中可以序列化 `TSerializer` 与反序列化 `TDeserializer` 构造函数指定具体的序列化协议。例如：

```

new TSerializer(new TBinaryProtocol.Factory())
new TDeserializer(new TBinaryProtocol.Factory())

```

Thrift 支持多种序列化协议，常用的有 `TBinaryProtocol`、`TCompactProtocol` 和 `TJSONProtocol`。其中 `TBinaryProtocol` 为二进制序列化协议，`TCompactProtocol` 可以看作 `TBinaryProtocol` 的升级版，采用了字节压缩算法，进一步减少了序列化后的码流，`TJSONProtocol` 是一种 JSON 数据格式序列化协议。

Maven 依赖配置：

```
<dependency>
  <groupId>org.apache.thrift</groupId>
  <artifactId>libfb303</artifactId>
  <version>0.9.3</version>
</dependency>
<dependency>
  <groupId>org.apache.thrift</groupId>
  <artifactId>libthrift</artifactId>
  <version>0.9.3</version>
</dependency>
```

3.9 Avro 序列化框架介绍

3.9.1 Avro 介绍

Avro 是一个数据序列化的项目，最开始是 Hadoop 的子项目之一，最后加入 Apache 成为独立的开源项目。Avro 提供的功能类似于其他编组系统，如 Thrift、protobuf 等。而 Avro 的主要不同之处在于以下几点。

- ◎ 动态类型：Avro 无须生成代码。数据总是伴以模式定义，这样就可以在不生成代码、静态数据类型的情况下对数据进行所有处理，有利于构建通用的数据处理系统和语言。
- ◎ 无标记数据：由于在读取数据时有模式定义，这就大大减少了数据编辑所需的类型信息，从而减少序列化空间开销。
- ◎ 不用手动分配的字段 ID：当数据模式发生变化，处理数据时总是同时提供新旧模式，差异就可以用字段名来做符号化的分析。

由于性能高、基本代码少和产出数据量精简等特点，很多知名开源项目都应用了 Avro，

包括 Hadoop、Cassandra 等。

Avro 具有如下特性。

- ◎ 丰富的数据结构类型。
- ◎ 快速可压缩的二进制数据形式。
- ◎ 存储持久数据的文件容器。
- ◎ 远程过程调用 RPC。
- ◎ 简单的动态语言结合功能，Avro 和动态语言结合后，读写数据文件和使用 RPC 协议都不需要生成代码，而代码生成作为一种可选的优化只值得在静态类型语言中实现。

Avro 支持两种序列化编码方式：二进制编码和 JSON 编码。使用二进制编码的性能更高，序列化后产生的码流更小，而使用 JSON 编码的好处是编码后的可读性好，适合在开发调试阶段使用。

3.9.2 Avro IDL 语言介绍

之前接触过 Thrift 与 protobuf 的读者应该对 IDL（接口描述语言）的概念不会感到陌生。与 Thrift、protobuf 类似，Avro IDL 接口描述语言，定义了 Avro 作为 RPC 框架时的 RPC 接口及 RPC 接口所需的数据结构。该定义文件以 .avdl 文件后缀为结尾，参考 <https://avro.apache.org/docs/current/idl.html>。

下面做一个简要介绍。

- ◎ Protocol：指定所生成的接口类类名。
- ◎ @namespace：指定所在的包名。
- ◎ @javaclass：指定字段对应的 Java 类型。

常用的数据类型如下。

- ◎ **enum**: 自定义枚举类型。
- ◎ **record**: 自定义数据传输结构，类似于 C 语言的结构体类型，与 Java Bean 对应。
- ◎ **string**: 字符串类型。
- ◎ **boolean**: 布尔类型。
- ◎ **int、long、float、double**: 数值类型。
- ◎ **array**: 数组类型。
- ◎ **map**: 散列数据结构。

下面给出一个综合性的示例：

```
@namespace("avro")//包名
protocol ExampleService { //接口类类名
    //定义枚举
    enum Suit {
        SPADES, DIAMONDS, CLUBS, HEARTS
    }
    //定义 record，类似于 C 语言结构体，可以看作 Java 语言的类
    record DataExample {
        string name; //字符串类型
        int age;      //int 类型
        boolean success; //bool 类型
        Suit suit; //自定义枚举类型
        @javaclass("java.util.ArrayList")array<string> addressList; //数组类型
        map<string> demoMap; //散列数据结构
    }

    //rpc 服务接口方法
    void sayHello(string content,DataExample data);
}
```

3.9.3 Schema 定义介绍

如果我们不将 Avro 作为 RPC 框架使用, 仅仅使用其数据序列化/反序列化的功能。则可以使用 JSON 格式来定义所需要序列化的数据结构 Schema, 定义文件以 .avsc 后缀结尾。整个 Schema 使用 JSON 结构来表示, 官方文档 <http://avro.apache.org/docs/1.7.7/spec.html>。

下面做一个简要介绍。

- ◎ namespace: 定义生成的代码所在的包路径。
- ◎ type: 指明当前的类型。
- ◎ name: 定义生成的类名。
- ◎ fields: 定义类包含的字段列表, 其中每一个 field 由 name 与 type 定义。

下面给出一个综合性示例:

```
{ "namespace": "avro",
  "type": "record",
  "name": "Demo",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "age", "type": [ "int", "null" ] },
    { "name": "dataList", "type": { "type": "array", "items": "long" } },
    { "name": "dataMap", "type": { "type": "map", "values": "string" } }
  ]
}
```

3.9.4 Maven 配置及使用 IDL 与 Schema 自动生成代码

在 Maven 增加如下配置:

```
<dependency>
```

```
<groupId>org.apache.avro</groupId>
<artifactId>avro</artifactId>
<version>1.8.1</version>
</dependency>
<dependency>
<groupId>org.apache.avro</groupId>
<artifactId>avro-ipc</artifactId>
<version>1.8.1</version>
</dependency>

<build>
<plugins>
.....
<plugin>
<groupId>org.apache.avro</groupId>
<artifactId>avro-maven-plugin</artifactId>
<version>1.8.1</version>
<executions>
<execution>
<phase>generate-sources</phase>
<goals>
<goal>schema</goal>
</goals>
<configuration>
<sourceDirectory>${project.basedir}/src/main/avro</sourceDirectory>
<outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
</configuration>
</execution>
</executions>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
```

```

<configuration>
<source>1.6</source>
<target>1.6</target>
</configuration>
</plugin>
<plugin>
<groupId>org.apache.avro</groupId>
<artifactId>avro-maven-plugin</artifactId>
<executions>
<execution>
<goals>
<goal>idl-protocol</goal>
</goals>
</execution>
</executions>
</plugin>
.....
</plugins>
</build>

```

在 pom.xml 所在目录下，在命令行中运行如下命令：mvn clean install，会自动生成相应的 RPC 接口类与数据传输的类，如图 3-2 所示。

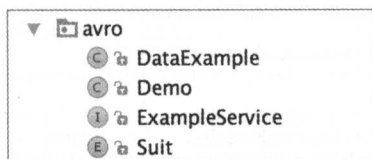


图 3-2 RPC 接口与数据传输的类

3.9.5 Avro 序列化/反序列化实现

序列化类 User.java 的 Schema 定义如下：

```

{"namespace": "avro",
 "type": "record",
 "name": "User",
 "fields": [
   {"name": "name", "type": "string"},
   {"name": "age", "type": ["int", "null"]},
   {"name": "email", "type": ["string", "null"]}
 ]
}

```

使用上述 Schema 生成 User.java 文件，使用预生成 User，序列化生成 byte[]，再反序列化生成新的 User 对象。

```

User userAvro = new User();
userAvro.setAge(28);
userAvro.setEmail("kongxuan@163.com");
userAvro.setName("kongxuan");
//1 序列化
DatumWriter<User> userDatumWriter = new SpecificDatumWriter<User>
(User.class);
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
BinaryEncoder binaryEncoder = EncoderFactory.get().directBinaryEncoder
(outputStream, null);
userDatumWriter.write(userAvro, binaryEncoder);
byte[] data = outputStream.toByteArray();
//2 反序列化
DatumReader<User> userDatumReader = new SpecificDatumReader<User>
(User.class);
BinaryDecoder binaryDecoder = DecoderFactory.get().directBinaryDecoder
(new ByteArrayInputStream(data), null);
User userAvroCopy = userDatumReader.read(new User(), binaryDecoder);

System.out.println("age:" + userAvroCopy.getAge() + " email:" +
userAvroCopy.getEmail() + " name:" + userAvroCopy.getName());

```

直接使用 schema 文件进行序列化与反序列化方法如下。

```
//根据 avsc 文件进行序列化的操作方式
Schema schema = new Schema.Parser().parse(new File("src/main/avro/user.
avsc"));

GenericRecord userAvro2 = new GenericData.Record(schema);
userAvro2.put("age", 18);
userAvro2.put("email", "liyebing@163.com");
userAvro2.put("name", "liyebing");

//1 序列化
DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter
<GenericRecord>(schema);
ByteArrayOutputStream outputStream2 = new ByteArrayOutputStream();
BinaryEncoder binaryEncoder2 = EncoderFactory.get().directBinaryEncoder
(outputStream2, null);
datumWriter.write(userAvro2, binaryEncoder2);
byte[] data2 = outputStream2.toByteArray();

//2 反序列化
DatumReader<User> userDatumReader2 = new SpecificDatumReader<User>
(User.class);
BinaryDecoder binaryDecoder2 = DecoderFactory.get().directBinaryDecoder
(new ByteArrayInputStream(data2), null);
User userAvroCopy2 = userDatumReader2.read(new User(), binaryDecoder2);
System.out.println("age:" + userAvroCopy2.getAge() + " email:" +
userAvroCopy2.getEmail() + " name:" + userAvroCopy2.getName());
```

将序列化的结果保存到文件中，代码如下。

```
//1 序列化
File file = new File("users.avro");
DataFileWriter<User> dataFileWriter = new DataFileWriter<User>
(userDatumWriter);
```

```
dataFileWriter.create(userAvro.getSchema(), file);
dataFileWriter.append(userAvro);
dataFileWriter.append(userAvroCopy2);
dataFileWriter.close();
```

//2 从文件中反序列化

```
DataFileReader<User> dataFileReader = new DataFileReader<User>(file,
userDatumReader);
User user = null;
while (dataFileReader.hasNext()) {
    user = dataFileReader.next(user);
    System.out.println("age:" + user.getAge() + " email:" + user.getEmail()
+ " name:" + user.getName());
}
```

下面抽取一个通用的 Avro 序列化/反序列化方法。

```
public class AvroSerializer implements ISerializer {

    public <T> byte[] serialize(T obj) {
        try {
            if (!(obj instanceof SpecificRecordBase)) {
                throw new UnsupportedOperationException("not supported obj
type");
            }
            DatumWriter userDatumWriter = new SpecificDatumWriter(obj.
getClass());
            ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
            BinaryEncoder binaryEncoder = EncoderFactory.get().
directBinaryEncoder(outputStream, null);
            userDatumWriter.write(obj, binaryEncoder);
            return outputStream.toByteArray();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

```

    }
}

public <T> T deserialize(byte[] data, Class<T> clazz) {
    try {
        if (!SpecificRecordBase.class.isAssignableFrom(clazz)) {
            throw new UnsupportedOperationException("not supported clazz
type");
        }
        DatumReader userDatumReader = new SpecificDatumReader(clazz);
        BinaryDecoder binaryDecoder = DecoderFactory.get().
directBinaryDecoder(new ByteArrayInputStream(data), null);
        return (T) userDatumReader.read(clazz.newInstance(),
binaryDecoder);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

Maven 依赖配置如下:

```

<dependency>
<groupId>org.apache.avro</groupId>
<artifactId>avro</artifactId>
<version>1.8.1</version>
</dependency>
<dependency>
<groupId>org.apache.avro</groupId>
<artifactId>avro-ipc</artifactId>
<version>1.8.1</version>
</dependency>

```

3.10 JBoss Marshalling 序列化框架介绍

JBoss Marshalling 是一个 Java 对象序列化包，兼容 Java 原生的序列化机制，对 Java 原生序列化机制做了优化，使其在性能上有很大提升。在保持跟 `java.io.Serializable` 接口兼容的同时增加了一些可调的参数和附加特性，这些参数和附加的特性，可通过工厂类进行配置，对原生 Java 序列化是一个很好的替代。JBoss Marshalling 序列化/反序列化示例如下：

```
public class MarshallingSerializer implements ISerializer {

    final static MarshallingConfiguration configuration = new
MarshallingConfiguration();

    //获取序列化工厂对象，参数 serial 标识创建的是 Java 序列化工厂对象
    final static MarshallerFactory marshallerFactory = Marshalling.
getProvidedMarshallerFactory("serial");

    static {
        configuration.setVersion(5);
    }

    public byte[] serialize(Object obj) {
        final ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
        try {
            final Marshaller marshaller = marshallerFactory.createMarshaller
(configuration);
            marshaller.start(Marshalling.createByteOutput
(byteArrayOutputStream));
            marshaller.writeObject(obj);
            marshaller.finish();
        }
```

```

    } catch (IOException e) {
        e.printStackTrace();
    }
    return byteArrayOutputStream.toByteArray();
}

public <T> T deserialize(byte[] data, Class<T> clazz) {
    try {
        ByteArrayInputStream byteArrayInputStream = new
        ByteArrayInputStream(data);
        final Unmarshaller unmarshaller = marshallerFactory.
        createUnmarshaller(configuration);
        unmarshaller.start(Marshalling.createByteInput
        (byteArrayInputStream));
        Object object = unmarshaller.readObject();
        unmarshaller.finish();
        return (T) object;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
}

```

Maven 依赖配置如下:

```

<dependency>
<groupId>org.jboss.marshalling</groupId>
<artifactId>jboss-marshalling-serial</artifactId>
<version>2.0.0.Beta2</version>
</dependency>

```

3.11 序列化框架的选型

前面我们已经介绍了 9 种最常用的的序列化框架的使用。每一种序列化协议都有自己的优缺点及适用场景。我们在选型的时候，一般会从下面几个方面来考察。

（1）技术层面的考虑

- ◎ 序列化空间开销，即序列化结果产生的码流大小，码流过大会对带宽、存储空间造成较大压力。
- ◎ 序列化时间开销，即序列化过程消耗的时长，序列化消耗时间过长会拖慢整个服务的响应时间。
- ◎ 序列化协议是否支持跨平台、跨语言，公司内部存在异构系统通信需求时，往往要求 RPC 框架采用的序列化协议支持跨平台、跨语言。
- ◎ 可扩展性/兼容性，在实际业务开发过程中，我们的系统往往需要随着需求的快速迭代快速更新，这就要求我们采用的序列化协议具有良好的可扩展性/兼容性，比如，在现有的序列化数据结构中新增某个业务字段，而不会影响到现有的业务服务。
- ◎ 成熟度及支持的数据结构的丰富性也是一个需要重点考量的方面。

（2）其他层面的考虑

- ◎ 技术的流行程度，背后是否有大公司技术支撑，是否是一个长期发展的持续进化的技术，是否已经得到业界的充分验证。
- ◎ 学习难度和易用性。

（3）选型建议

- ◎ 对于公司间的系统调用，性能要求在 100ms 以上的服务，基于 XML 的 SOAP 协

议是一个值得考虑的方案。

- ◎ 基于 Web Browser 的 Ajax, 以及 Mobile App 与服务端之间的通信, JSON 协议是首选。对于性能要求不太高, 或者以动态类型语言为主, 或者传输数据载荷很小的运用场景, JSON 也是一个非常不错的选择。
- ◎ 对于调试环境比较恶劣的场景, 采用 JSON 或 XML 能够极大地提高调试效率, 降低系统开发成本。
- ◎ 对性能和简洁性有极高要求的场景, Hessian、protobuf、Thrift、Avro 之间具有一定的竞争关系。其中 Hessian 是在性能和稳定性同时考虑下最优的序列化协议。
- ◎ 对于 T 级别的数据的持久化应用场景, protobuf 和 Avro 是首要选择。如果持久化后的数据存储在 Hadoop 子项目里, Avro 会是更好的选择。
- ◎ 由于 Avro 的设计理念偏向于动态类型语言, 对于以动态语言为主的应用场景, Avro 是更好的选择。
- ◎ 对于持久层非 Hadoop 项目, 以静态类型语言为主的应用场景, protobuf 会更符合静态类型语言工程师的开发习惯。
- ◎ 对需要提供一个完整的 RPC 解决方案, Thrift 是一个好的选择。
- ◎ 对序列化之后需要支持不同的传输层协议, 或者需要跨防火墙访问的高性能场景, Protobuf 可以优先考虑^①。

3.12 实现自己的序列化工具引擎

本节将整合之前介绍的 9 种序列化解决方案。前文已经给出下面这 9 种序列化方式使用的通用方法:

^① 参考 http://tech.meituan.com/serialization_vs_deserialization.html。

```
DefaultJavaSerializer.java
XmlSerializer.java
JSONSerializer.java
HessianSerializer.java
ProtoBufSerializer.java
ProtoStuffSerializer.java
ThriftSerializer.java
AvroSerializer.java
MarshallngSerializer.java
```

下面将依据以上的实现，整合为通用的序列化工具引擎，可以通过输入不同配置随意选择使用哪一种序列化方案。该序列化工具引擎最终会应用在本书完成的分布式服务框架里面。

```
import com.google.common.collect.Maps;
import serializer.ISerializer;
import serializer.SerializeType;
import serializer.impl.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class SerializerEngine {

    public static final Map<SerializeType, ISerializer> serializerMap =
Maps.newConcurrentMap();

    //注册序列化工具类到 serializerMap
    static {
        serializerMap.put(SerializeType.DefaultJavaSerializer, new
DefaultJavaSerializer());
        serializerMap.put(SerializeType.HessianSerializer, new
HessianSerializer());
```



```

        serializerMap.put(SerializeType.JSONSerializer, new
JSONSerializer());
        serializerMap.put(SerializeType.XmlSerializer, new
XmlSerializer());
        serializerMap.put(SerializeType.ProtoStuffSerializer, new
ProtoStuffSerializer());
        serializerMap.put(SerializeType.MarshallingSerializer, new
MarshallingSerializer());

        //以下三类不能使用普通的 Java Bean, 需要根据各自 IDL 编译生成的类
        serializerMap.put(SerializeType.AvroSerializer, new
AvroSerializer());
        serializerMap.put(SerializeType.ThriftSerializer, new
ThriftSerializer());
        serializerMap.put(SerializeType.ProtocolBufferSerializer, new
ProtoBufSerializer());
    }

    /**
     * 序列化
     *
     * @param obj
     * @param serializeType
     * @param <T>
     * @return
     */
    public static <T> byte[] serialize(T obj, String serializeType) {
        SerializeType serialize = SerializeType.queryByType(serializeType);
        if (serialize == null) {
            throw new RuntimeException("serialize is null");
        }
    }

```

```
        ISerializer serializer = serializerMap.get(serialize);
        if (serializer == null) {
            throw new RuntimeException("serialize error");
        }

        try {
            return serializer.serialize(obj);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * 反序列化
     *
     * @param data
     * @param clazz
     * @param serializeType
     * @param <T>
     * @return
     */
    public static <T> T deserialize(byte[] data, Class<T> clazz, String
serializeType) {

        SerializeType serialize = SerializeType.queryByType(serializeType);
        if (serialize == null) {
            throw new RuntimeException("serialize is null");
        }

        ISerializer serializer = serializerMap.get(serialize);
        if (serializer == null) {
            throw new RuntimeException("serialize error");
        }
    }
}
```

```

    try {
        return serializer.deserialize(data, clazz);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

其中，`SerializeType` 是序列化类型枚举：

```

import org.apache.commons.lang.StringUtils;

public enum SerializeType {
    DefaultJavaSerializer("DefaultJavaSerializer"),
    HessianSerializer("HessianSerializer"),
    JsonSerializer("JsonSerializer"),
    ProtoStuffSerializer("ProtoStuffSerializer"),
    XmlSerializer("XmlSerializer"),
    MarshallingSerializer("MarshallingSerializer"),
    AvroSerializer("AvroSerializer"),
    ProtocolBufferSerializer("ProtocolBufferSerializer"),
    ThriftSerializer("ThriftSerializer");

    private String serializeType;

    private SerializeType(String serializeType) {
        this.serializeType = serializeType;
    }

    public static SerializeType queryByType(String serializeType) {
        if (StringUtils.isBlank(serializeType)) {
            return null;
        }
        for (SerializeType serialize : SerializeType.values()) {

```

```

        if (StringUtils.equals(serializeType, serialize.getSerializeType())) {
            return serialize;
        }
    }
    return null;
}

public String getSerializeType() {
    return serializeType;
}
}

```

序列化引擎 `SerializerEngine` 通过 `static` 块，在类加载的时候，将 9 类序列化算法注册到本地缓存 `serializerMap` 中，提供了相应的序列化与反序列化的通用处理方法。

- ◎ 序列化方法 `public static <T> byte[] serialize(T obj, String serializeType)`，该方法传入参数为待序列化对象，以及序列化方式。
- ◎ 反序列化方法 `public static <T> T deserialize(byte[] data, Class<T> clazz, String serializeType)`，该方法传入参数为字节数组、反序列化结果对象类型，以及序列化方式。

通过 `SerializerEngine` 引擎，可以通过传入参数 `serializeType` 的方式灵活选择具体的序列化/反序列化方案，做到序列化/反序列化方案的可配置化。

3.13 本章小结

本章概要性地介绍了序列化原理，对常见的 9 种序列化方式做了总结，并给出了各自的序列化/反序列化实现。同时对各自序列化方案的使用场景及如何选型给了相应的建议。最后，通过将以上 9 种序列化/反序列化方案整合，实现了可配置化的序列化引擎 `SerializerEngine`，该引擎将在之后的分布式服务框架中使用。

第 4 章

实现分布式服务框架服务的 发布与引入

4.1 Spring Framework 框架概述

4.1.1 Spring Framework 介绍

Spring Framework 为 Java 企业级开发提供了一站式的轻量级（相对 EJB 而言）解决方案，目前已经成为 Java 企业级开发领域事实上的标准。

Spring 抽象了我们在开发过程中遇到的很多共性的问题，为我们解决这些问题提供了很好的脚手架。

◎ 提供了编程式事务模板类 `TransactionTemplate.java` 与声明式事务注解 `@Transactional`

的解决方案，简化了开发过程中事务控制的繁杂工作。

- ◎ 以 `DataAccessException.java` 为基类，抽象了统一的数据库异常表示。
- ◎ 提供了统一的数据库集成抽象层，同时通过提供模板类 `JdbcTemplate.java` 简化了 JDBC 操作代码。
- ◎ 提供了 Spring MVC 这一优秀的 MVC 框架，极大简化了开发人员在展现层与后台服务调用之间的工作，同时也提供了扩展点，可以无缝集成现有的其他主流 MVC 框架（Struts1.x、Struts2.x、WebWork 等），实际上，Spring MVC 渐渐有取代其他 MVC 框架的趋势。
- ◎ 提供了 Spring AOP 及通过集成 Aspectj 为 AOP 开发提供了开箱即用的强大支持。
- ◎ 最核心的是提供了 IOC 容器，提供了依赖反转模式的实现，为 Java 企业级开发带来了革命性的创新体验。

Spring 还有大量其他的有用的特性，限于篇幅，不在此一一列举了。同时，Spring 本身也是基于模块化构建的，在实际使用的时候，可以按需引入所需的模块，如图 4-1 所示。

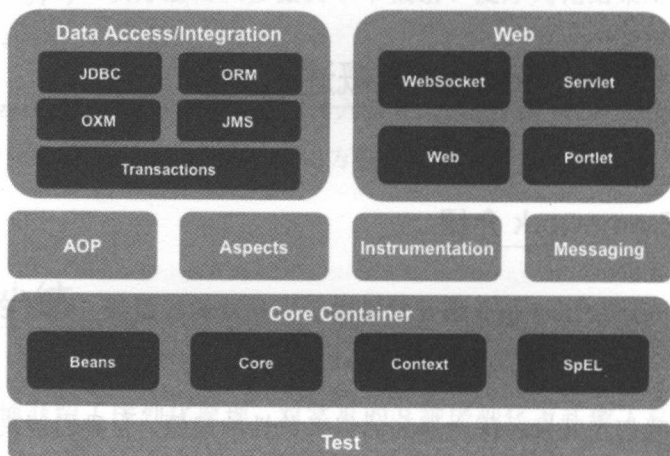


图 4-1 Spring Runtime 架构图

4.1.2 Spring Framework 周边生态项目介绍

此外，围绕 Spring Framework 周边衍生了一批高质量的开源项目，能够与 Spring Framework 很好地集成，为某个领域提供特定的解决方案。

- ◎ **Spring I/O Platform:** 为解决开发过程中依赖过于复杂导致的版本不一致问题所提供的解决方案。基于 Maven 的 dependencyManagement 功能声明了常用的软件包的版本，所提供的版本之间是互相兼容而且无冲突的，开发人员只需通过 Maven 继承 Spring I/O Platform 所声明的版本列表，可免除需要手工确定所依赖的软件包的版本的繁杂工作。官网为 <http://platform.spring.io/platform>。
- ◎ **Spring Boot:** 该项目的目的是为了将开发人员从 Spring 开发过程中繁杂的配置工作中解放出来。Spring Boot 基于约定优于配置的理念，极大地简化了利用 Spring 开发所需要进行的各种配置工作，几乎可以做到零配置。而且是最近兴起的微服务概念实际落地的常用技术。官网为 <http://projects.spring.io/spring-boot/>。
- ◎ **Spring Data:** 该项目的使命是为数据访问提供一个熟悉且一致的基于 Spring 的编程模型，同时仍然保留基础数据存储的特殊特性。官网为 <http://projects.spring.io/spring-data/>。
- ◎ **Spring Cloud:** 该项目为开发人员提供了快速构建分布式系统中的一些通用模型（例如配置管理、服务发现、智能路由、服务代理、控制总线、一次性令牌、全局锁、选举、分布式会话等）。开发人员可以基于 Spring Cloud 快速实现该模型的服务。官网为 <http://projects.spring.io/spring-cloud/>。
- ◎ **Spring Integration:** 该项目的主要目标是提供一个用于构建企业集成解决方案的简单模型。通过扩展 Spring 编程模型以支持众所周知的企业集成模式。在基于 Spring 的应用程序中实现轻量级消息传递并支持通过声明适配器与外部系统集成。这些适配器提供了比 Spring 对远程、消息和调度的支持更高级的抽象。官网为 <http://projects.spring.io/spring-integration/>。
- ◎ **Spring Batch:** 轻量级、全面的批处理框架。为了企业级开发中批处理应用提供了

强大的解决方案。官网为 <http://projects.spring.io/spring-batch/>。

- ◎ **Spring Security:** Spring Security 是一个提供对 Java 应用程序的认证和授权的框架。Spring Security 的强大之处在于提供了大量的扩展点，可以被轻松地扩展以满足定制需求。官网为 <http://projects.spring.io/spring-security/>。
- ◎ **Spring Web Services:** Spring-WS 致力于快速开发契约优先(Contract First)的 SOAP 服务，并使用多种操作 XML 有效负荷的方式来创建灵活的 WebService。这款产品基于 Spring 自身，这意味着我们可以使用依赖注入等 Spring 的概念来整合 WebService 服务。Spring-WS 使得 WebService 好用却十分容易上手。Spring-WS 包含了 WS-I 的基本描述、契约优先的开发模式、在契约和实现之间的松耦合实现等。官网为 <http://projects.spring.io/spring-ws/>。
- ◎ **Spring Session:** 该项目为分布式 Session 的实现提供了很好的脚手架。官网为 <http://projects.spring.io/spring-session/>。
- ◎ **Spring For Android:** 该项目是 Spring Framework 的扩展，旨在简化原生 Android 应用程序的开发。官网为 <http://projects.spring.io/spring-android/>。

以上，我们从整体上介绍了 Spring Framework 及 Spring Framework 周边衍生的开发生态环境。

4.2 FactoryBean 的秘密

鉴于 Spring 在 Java 企业级开发中的地位，我们希望分布式服务框架能够与 Spring 无缝地集成，远程服务的发布与引入与本地服务的发布与引入对于开发人员的编程界面能够保持一致。本节重点介绍 Spring 中的 FactoryBean 接口，解释该接口的实现及运行原理与作用，使读者理解后续如何使用该接口实现远程服务的发布与引入。

4.2.1 FactoryBean 的作用及使用场景

FactoryBean 接口代码如下：

```
public interface FactoryBean<T> {  
    T getObject() throws Exception;  
    Class<?> getObjectType();  
    boolean isSingleton();  
}
```

其中：

- ◎ T getObject()方法负责返回 Java Bean 的具体实例对象，如果 isSingleton()返回 true，则该实例会放到 Spring 容器中单实例缓存池中。
- ◎ Class<?>getObjectType()方法负责返回 Java Bean 类型。
- ◎ Boolean isSingleton()方法负责说明返回的 Java Bean 对象实例是否是单例对象。

我们通过 Spring 框架 IOC 容器获取一个 Bean 实例，一般的做法是通过 XML 配置（也可以通过注解@Component、@Controller、@Service 等），比如：

```
<bean id="user" class="demo.model.User">  
    <property name="name" value="kongxuan"/>  
    <property name="email" value="liyebing@163.com"/>  
</bean>
```

实现原理是 Spring 通过反射机制利用 Bean 的 class 实例化 Bean。在某些情况下，实例化 Bean 过程如果涉及复杂的业务逻辑，通过 XML 配置或者注解的方式实例化这样一个对象很难实现或者实现起来不够灵活。这个时候，通过实现 org.springframework.beans.factory.FactoryBean 接口，采用编码的方式来实例化一个 Bean，将实例化 Bean 相关的复杂业务逻辑通过编码在方法 org.springframework.beans.factory.FactoryBean#getObject 中来实现，这可能是一个相对简单的方案。

4.2.2 FactoryBean 实现原理及示例说明

通过 FactoryBean 来实例化 Bean 的原理是什么呢？下面我们通过查看 Spring 的源码来一探究竟。通过查看在 Spring 中获取一个 Bean 实例的代码执行流程，可以发现如下执行流程，如图 4-2 所示。

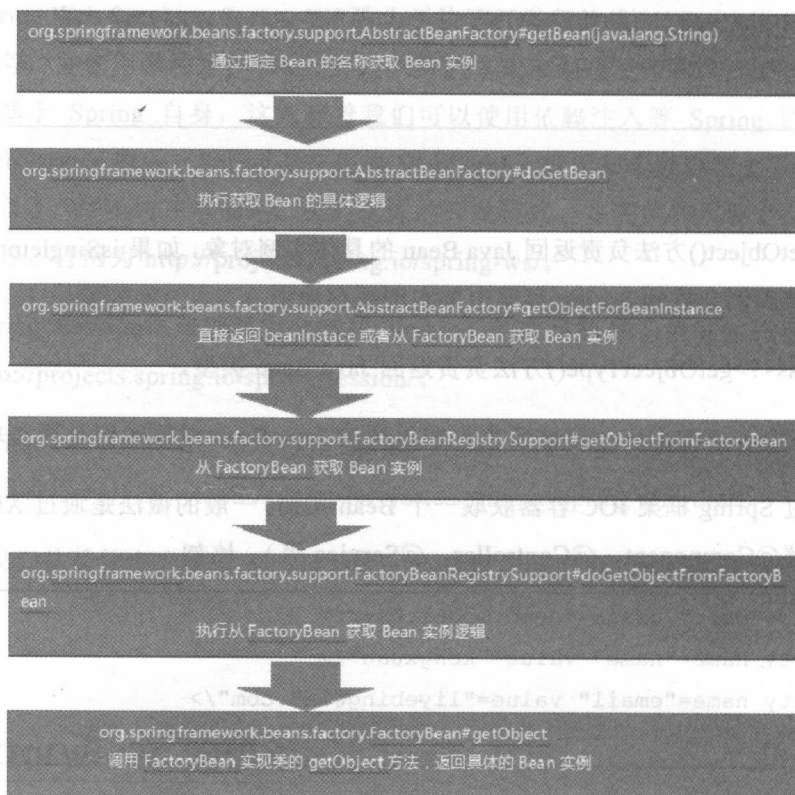


图 4-2 Bean 实例的代码执行流程

下面通过一个完整的示例来说明如何通过 FactoryBean 接口来实例化我们所需要的 Bean。

```

public class UserFactoryBean implements FactoryBean<User> {

    private static final User user = new User();
  
```

```
private String name;
private String email;

public User getObject() throws Exception {
    //这里可以实现复杂业务逻辑,这里仅仅是演示代码,逻辑很简单
    user.setName(name);
    user.setEmail(email);
    return user;
}

public Class<?> getObjectType() {
    return User.class;
}

public boolean isSingleton() {
    return true;
}

public void setName(String name) {
    this.name = name;
}

public void setEmail(String email) {
    this.email = email;
}
}
```

Spring 配置文件内容如下:

```
<bean id = "user" class="demo.factorybean.UserFactoryBean">
<property name="name" value="kongxuan"/>
<property name="email" value="liyebing@163.com"/>
</bean>
</beans>
```

通过如下方式获取 User 对象:

```
public class MainClient {  
    public static void main(String[] args) {  
        //加载 spring 配置, 启动 Spring 运行时  
        ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");  
  
        //获取通过 UserFactoryBean 生产的 User 对象  
        User user = (User) context.getBean("user");  
        System.out.println(user.getEmail() + " " + user.getName());  
    }  
}
```

整个工程结构如图 4-3 所示。

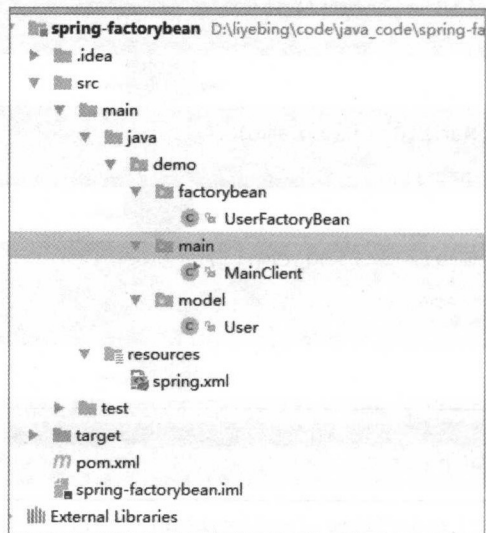


图 4-3 FactoryBean 示例工程结构

以上代码通过实现 `demo.factorybean.UserFactoryBean.java` 来实例化 `User` 实例对象。其中方法 `public User getObject()` 里面可以用来实现我们的实例化 `User` 对象的逻辑。

4.3 Spring 框架对于已有 RPC 框架集成的支持

4.3.1 Spring 支持集成 RPC 框架介绍

Spring 对常见的 RPC 框架提供了相应的集成支持，统一标准化了 RPC 服务发布与引入编程模型，简化了 RPC 服务及调用的开发流程。目前 Spring 主要支持如下的远程调用技术。

- ◎ Remote Method Invocation (RMI): 通过类 `RmiProxyFactoryBean` 与类 `RmiServiceExporter` 分别完成 RMI 服务的引入与发布。
- ◎ HTTP Invoker: Spring 提供了一种远程调用技术，使用 Java 内置的序列化算法，以及通过 HTTP 协议进行数据的传输。相应的支持类是 `HttpInvokerProxyFactoryBean` 和 `HttpInvokerServiceExporter`。与 RMI 的相同点在于数据序列化算法相同，都是采用 Java 内置的序列化算法。不同点在于 RMI 传输协议为 TCP/IP 协议，而 HTTP Invoker 为 HTTP 协议。HTTP 协议传输性能低于 TCP/IP 协议，但是不会被防火墙拦截。
- ◎ Hessian: Hessian 是一个轻量级的 Web 服务实现工具，它采用的是二进制协议，因此很适合发送二进制数据。它的一个基本原理就是把远程服务对象以二进制的方式进行发送和接收。Spring 通过 `HessianProxyFactoryBean` 和 `HessianServiceExporter` 来完成基于 Hessian 序列化协议的服务的引入与发布。
- ◎ JAX-WS: Spring 通过 JAX-WS (Java EE 5 和 Java 6 中引入的 JAX-RPC 的处理器) 提供对 WS 服务的支持。类似地提供了相应的支持类 `JaxWsPortProxyFactoryBean` 与 `SimpleJaxWsServiceExporter` 来完成 WS 服务的引入与发布。
- ◎ JMS: 相应的支持类为 `JmsInvokerProxyFactoryBean` 与 `JmsInvokerServiceExporter`。

通过查看源码，可以发现 `RmiProxyFactoryBean`、`HttpInvokerProxyFactoryBean`、

HessianProxyFactoryBean、JaxWsPortProxyFactoryBean、JmsInvokerProxyFactoryBean 均实现了接口 `org.springframework.beans.factory.FactoryBean`。

下面通过具体的示例分别介绍 RMI、HttpInvoker、Hessian 三种 RPC 框架如何与 Spring 集成。

4.3.2 基于 RmiProxyFactoryBean 实现 RMI 与 Spring 的集成

Spring 通过 `org.springframework.remoting.rmi.RmiProxyFactoryBean` 引入 RMI 服务，通过 `org.springframework.remoting.rmi.RmiServiceExporter` 暴露 RMI 服务，屏蔽掉了 RMI 开发的复杂性，对比 RMI 原生开发，不用关心服务端 Skeleton 和客户端 Stub 等的处理细节，其远程服务接口与实现类甚至不用实现 `java.rmi.Remote` 接口与继承类 `java.rmi.server.UnicastRemoteObject`，简单服务接口与实现类代码如下。

```
public interface UserService {
    public User findByName(String userName);
}

@Service("userService")
public class UserServiceImpl implements UserService {
    private static final Map<String, User> userMap = new HashMap<String,
User>();

    static {
        userMap.put("kongxuan", new User("kongxuan", "kongxuan@163.com"));
        userMap.put("liyebing", new User("liyebing", "liyebing@163.com"));
    }

    public User findByName(String userName) {
        return userMap.get(userName);
    }
}
```

```

public class User implements Serializable {
    private String name;
    private String email;

    public User(){}
    public User(String name,String email){
        this.name =name;
        this.email=email;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}

```

RMI 服务发布与引入的配置文件如下，rmi-rpc-server.xml 实现 RMI 服务的发布。

```

<!-- 将 userService 暴露为远程服务 -->
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
<!-- 这里的服务名必须和客户端调用协议 rmi://hostname:1199/xxxxxxx 的 xxxxxxx 相同 -->
-->
<property name="serviceName" value="userRmiService"/>
<property name="service" ref="userService"/>
<property name="serviceInterface" value="rpc.common.UserService"/>
<!-- 端口号，默认为 1099 -->

```

```
<property name="registryPort" value="1199"/>
</bean>
```

至此，服务端的代码已经完成。

rmi-rpc-client.xml 实现客户端 RMI 服务的引入，主体内容如下：

```
<!-- 客户端调用远程服务 -->
<bean
    id="userRmiServiceProxy"      class="org.springframework.remoting.rmi.
RmiProxyFactoryBean">
    <!-- 接收的 rmi 协议 -->
    <property name="serviceUrl" value="rmi://127.0.0.1:1199/userRmiService"/>
    <!-- 接收的 rmi 协议的接口 -->
    <property name="serviceInterface" value="rpc.common.UserService"/>
</bean>
```

通过 RmiInvokerClient 实现客户端调用 RMI 服务：

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import rpc.common.User;
import rpc.common.UserService;

public class RmiInvokerClient {

    public static void main(String[] args) {
        ApplicationContext context =
new ClassPathXmlApplicationContext("rmi-rpc-client.xml");
        UserService userService =
(UserService) context.getBean("userRmiServiceProxy");
        User user = userService.findByName("kongxuan");
        System.out.println(user.getName() + " " + user.getEmail());
    }
}
```

```
}
```

先将服务端部署启动，然后运行 RmiInvokerClient，得到运行结果如下：

```
kongxuan kongxuan@163.com
```

4.3.3 基于 HttpInvokerProxyFactoryBean 实现 HTTP Invoker 与 Spring 的集成

HTTP Invoker 作为 Spring 提供的一种新的 RPC 解决方案，其目的是为了填补 RMI 服务与基于 HTTP 服务（如 Hessian 等）之间的空白。因为 RMI 服务容易被防火墙拦截，Hessian 的序列化协议是私有协议。而 HTTP Invoker 采用与 RMI 相同的序列化协议（Java 内置序列化协议），传输采用 HTTP 协议，可以不被防火墙拦截，代码如下。

```
<import resource="common-config.xml"/>

<!-- 将 userService 暴露为远程服务 -->
<bean
    id="serviceHttpInvokeExporter" class="org.springframework.remoting.
httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="userService" />
    <property name="serviceInterface" value="rpc.common.UserService" />
</bean>
```

其中 common-config.xml 实现了 HTTP 服务路由配置：

```
<!-- Spring 注解扫描路径配置 -->
<context:component-scan base-package="rpc.*"/>

<!-- 映射 -->
<bean
    id="urlMapping" class="org.springframework.web.servlet.handler.
SimpleUrlHandlerMapping">
```

```
<property name="mappings">
<props>
<!-- httpInvoker -->
<prop key="/user.httpInvoker">serviceHttpInvokeExporter</prop>
</props>
</property>
</bean>
```

web.xml 关键配置代码如下：

```
<servlet>
<servlet-name>dispatcherServlet</servlet-name>
<servlet-class>
    org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<init-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:*-server.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>dispatcherServlet</servlet-name>
<url-pattern>*.httpInvoker</url-pattern>
</servlet-mapping>
```

至此服务端代码已经完成。

客户端配置 httpinvoker-rpc-client.xml 实现 HttpInvoker 服务的引入：

```
<!-- 客户端调用远程服务 -->
<bean
    id="userServiceProxy" class="org.springframework.remoting.httpinvoker.
HttpInvokerProxyFactoryBean">
    <property
```

```
name="serviceUrl" value="http://127.0.0.1:8080/user.httpInvoker"/>
<property name="serviceInterface" value="rpc.common.UserService"/>
</bean>
```

HttpInvokeClient 服务调用端:

```
import rpc.common.User;
import rpc.common.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplication
Context;

public class HttpInvokeClient {
    public static void main(String[] args) {
        ApplicationContext context =
        new ClassPathXmlApplicationContext("httpinvoker-rpc-client.xml");
        UserService userService =
        (UserService) context.getBean("userServiceProxy");

        User user = userService.findByName("liyebing");
        System.out.println(user.getName() + " " + user.getEmail());
    }
}
```

先将服务端部署启动起来，然后运行 HttpInvokeClient，得到运行结果如下：

```
liyebing    liyebing@163.com
```

4.3.4 基于 HessianProxyFactoryBean 实现 Hessian 与 Spring 的集成

Hessian 是一个轻量级的 remoting onhttp 工具，相比 WebService，Hessian 更简单、快捷。采用的是二进制 RPC 协议，因为采用的是二进制协议，所以它很适合发送二进制数据。下面通过具体的代码示例来演示在 Spring 框架中如何发布与引入 Hessian RPC 服务。

配置文件 `hessian-rpc-server.xml` 实现了 Hessian 服务的发布，关键配置如下：

```
<beans>
  <!-- 将 userService 暴露为远程服务 -->
  <bean
    name="serviceHessianExport"      class="org.springframework.remoting.
caucho.HessianServiceExporter">
    <property name="service" ref="userService" />
    <property name="serviceInterface" value="rpc.common.UserService" />
  </bean>
</beans>
```

配置文件 `common-config.xml` 实现了 Hessian HTTP 服务路由配置，关键配置如下：

```
<!-- 映射 -->
<bean
  id="urlMapping"      class="org.springframework.web.servlet.handler.
SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <!-- hessian -->
      <prop key="/user.hessianInvoker">serviceHessianExport</prop>
    </props>
  </property>
</bean>
```

`web.xml` 关键配置如下：

```
<servlet>
  <servlet-name>dispatcherServlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
```

```

<param-value>classpath:*-server.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>dispatcherServlet</servlet-name>
<url-pattern>*.hessianInvoker</url-pattern>
</servlet-mapping>

```

至此，服务端代码已经完成。

客户端配置 **hessian-rpc-client.xml** 实现服务的引入的关键部分如下：

```

<beans>
  <!-- 客户端调用远程服务 -->
  <beanid="userServiceProxy"          class="org.springframework.remoting.
httpInvoker.HttpInvokerProxyFactoryBean">
    <property
      name="serviceUrl" value="http://127.0.0.1:8080/user.httpInvoker"/>
    <property name="serviceInterface" value="rpc.common.UserService"/>
  </bean>
</beans>

```

客户端 **HessianInvokerClient** 服务调用端代码如下：

```

public class HessianInvokerClient {
    public static void main(String[] args) {
        ApplicationContext context
        = new ClassPathXmlApplicationContext("hessian-rpc-client.xml");
        UserService userService
        = (UserService) context.getBean("userServiceHessianProxy");
        User user = userService.findByName("kongxuan");
        System.out.println(user.getName() + " " + user.getEmail());
    }
}

```

先将服务端部署启动起来，然后运行 `HessianInvokerClient`，得到运行结果如下：

```
kongxuan    kongxuan@163.com
```

4.4 实现自定义服务框架与 Spring 的集成

本节实现分布式服务框架与 Spring 的集成，使用 Spring 管理远程服务的发布与引入。服务的发布与引入通过 Spring 管理，进而远程服务发布 Bean 与远程服务引入 Bean 通过 Spring IOC 容器管理，由 Spring 管理其生命周期。实现了远程服务调用编程界面与本地 Bean 方法调用的一致性，屏蔽了远程服务调用与本地方法调用的差异性。

4.4.1 实现远程服务的发布

远程服务自身的属性定义如下：

```
//服务接口
private Class<?> serviceItf;

//服务实现
private Object serviceObject;

//服务端口
private String serverPort;

//服务超时时间
private long timeout;

//服务代理对象，暂时没有用到
private Object serviceProxyObject;

//服务提供者唯一标识
private String appKey;

//服务分组组名
private String groupName = "default";

//服务提供者权重，默认为 1，范围为[1-100]
```

```
private int weight = 1;
//服务端线程数，默认10个线程，实现服务端流控
private int workerThreads = 10;
```

远程服务发布需要完成如下操作，具体见代码如下。

◎ 启动 Netty 服务端。

◎ 启动 ZooKeeper 服务端，将服务提供者属性信息注册到服务注册中心。

```
1  import ares.remoting.framework.Helper.IPHelper;
2  import ares.remoting.framework.model.ProviderService;
3  import ares.remoting.framework.zookeeper.IRegisterCenter4Provider;
4  import ares.remoting.framework.zookeeper.RegisterCenter;
5  import com.google.common.collect.Lists;
6  import org.springframework.beans.factory.FactoryBean;
7  import org.springframework.beans.factory.InitializingBean;
8  import java.lang.reflect.Method;
9  import java.util.List;
10
11 public class ProviderFactoryBean implements FactoryBean,
InitializingBean {
12     //服务接口
13     private Class<?> serviceItf;
14     //服务实现
15     private Object serviceObject;
16     //服务端口
17     private String serverPort;
18     //服务超时时间
19     private long timeout;
20     //服务代理对象，暂时没有用到
21     private Object serviceProxyObject;
22     //服务提供者唯一标识
23     private String appKey;
```

```
24     //服务分组组名
25     private String groupName = "default";
26     //服务提供者权重，默认为1，范围为[1-100]
27     private int weight = 1;
28     //服务端线程数，默认10个线程
29     private int workerThreads = 10;
30
31     @Override
32     public Object getObject() throws Exception {
33         return serviceProxyObject;
34     }
35
36     @Override
37     public Class<?> getObjectType() {
38         return serviceItf;
39     }
40
41     @Override
42     public boolean isSingleton() {
43         return true;
44     }
45
46
47     @Override
48     public void afterPropertiesSet() throws Exception {
49         //启动 Netty 服务端
50         NettyServer.singleton().start(Integer.parseInt(serverPort));
51
52         //注册到 ZooKeeper，元数据注册中心
53         List<ProviderService> providerServiceList =
buildProviderServiceInfos();
54         IRegisterCenter4Provider registerCenter4Provider =
55         RegisterCenter.singleton();
```



```

56
registerCenter4Provider.registerProvider(providerServiceList);
57     }
58
59
60     private List<ProviderService> buildProviderServiceInfos() {
61         List<ProviderService> providerList = Lists.newArrayList();
62         Method[] methods =
serviceObject.getClass().getDeclaredMethods();
63         for (Method method : methods) {
64             ProviderService providerService = new ProviderService();
65             providerService.setServiceItf(serviceItf);
66             providerService.setServiceObject(serviceObject);
67             providerService.setServerIp(IPHelper.localIp());
68
providerService.setServerPort(Integer.parseInt(serverPort));
69             providerService.setTimeout(timeout);
70             providerService.setServiceMethod(method);
71             providerService.setWeight(weight);
72             providerService.setWorkerThreads(workerThreads);
73             providerService.setAppKey(appKey);
74             providerService.setGroupName(groupName);
75             providerList.add(providerService);
76         }
77         return providerList;
78     }
79
80
81     public Class<?> getServiceItf() {
82         return serviceItf;
83     }
84
85     public void setServiceItf(Class<?> serviceItf) {

```

```
86         this.serviceItf = serviceItf;
87     }
88
89     public Object getServiceObject() {
90         return serviceObject;
91     }
92
93     public void setServiceObject(Object serviceObject) {
94         this.serviceObject = serviceObject;
95     }
96
97     public String getServerPort() {
98         return serverPort;
99     }
100
101     public void setServerPort(String serverPort) {
102         this.serverPort = serverPort;
103     }
104
105     public long getTimeout() {
106         return timeout;
107     }
108
109     public void setTimeout(long timeout) {
110         this.timeout = timeout;
111     }
112
113     public Object getServiceProxyObject() {
114         return serviceProxyObject;
115     }
116
117     public void setServiceProxyObject(Object serviceProxyObject) {
118         this.serviceProxyObject = serviceProxyObject;
```

```
119     }
120
121     public int getWeight() {
122         return weight;
123     }
124
125     public void setWeight(int weight) {
126         this.weight = weight;
127     }
128
129     public int getWorkerThreads() {
130         return workerThreads;
131     }
132
133     public void setWorkerThreads(int workerThreads) {
134         this.workerThreads = workerThreads;
135     }
136
137     public String getAppKey() {
138         return appKey;
139     }
140
141     public void setAppKey(String appKey) {
142         this.appKey = appKey;
143     }
144
145     public String getGroupName() {
146         return groupName;
147     }
148
149     public void setGroupName(String groupName) {
150         this.groupName = groupName;
151     }
```

其中第 13~30 行代码定义了远程服务发布相对应的属性，包括以下内容。

- ◎ 服务接口 `class (serviceItf)`：用于注册在服务注册中心，服务调用端获取后换成在本地缓存，用于发起服务调用。
- ◎ 服务实现 `Bean (serviceObject)`：用于服务调用。
- ◎ 服务启动端口 (`serverPort`)：对外发布服务作为 Netty 服务端端口。
- ◎ 服务端服务超时时间 (`timeout`)：用于控制服务端运行超时时间。
- ◎ 服务提供者唯一标识 (`appKey`)：唯一标识服务所在应用，作为 ZooKeeper 服务注册路径中的子路径，用于该应用所有服务的一个命名空间。
- ◎ 服务分组组名 (`groupName`)：用于分组灰度发布，比如某个服务 A，通过配置不同的分组组名，可以使得调用端发起的调用只路由到与其配置的分组组名相同的服务提供者机器组上。
- ◎ 服务提供者权重 (`weight`)：配置该机器对外发布的服务在集群中的权重，用于负载均衡相关的权重算法实现。
- ◎ 服务端线程数 (`workerThreads`)：限制服务端该服务运行线程数，用于实现资源的隔离与服务端限流。

第 49~58 行 `afterPropertiesSet()` 方法是接口 `org.springframework.beans.factory.InitializingBean` 中定义的方法。该接口会在 Spring Bean 初始化的时候自动执行一次，我们一般会使用该方法来初始化某些资源。我们主要做了两件事情：第一，调用 `NettyServer.singleton().start()` 方法来启动 Netty 服务端，将服务对外发布出去，使其能够接受外部其他机器的调用请求；第二，将服务信息写入 ZooKeeper，保存在服务注册中心。方法 `buildProviderServiceInfos()` 将服务接口按照方法的粒度拆分，获得服务方法粒度的服务列表 `List<ProviderService>`，然后通过调用注册中心的方法 `registerCenter4Provider.registerProvider()` 完成服务端信息的注册。

为了实际演示远程服务的发布，我们在服务端实现如下接口及接口实现类：

```
package com.ares.remoting.test

public interface HelloService {

    public String sayHello(String somebody);

}

package com.ares.remoting.test

public class HelloServiceImpl implements HelloService {

    @Override

    public String sayHello(String somebody) {

        return "hello " + somebody + "!";

    }

}
```

我们通过下面的配置将服务 `HelloService` 作为远程服务发布出去，下面是发布远程服务的 XML 配置：

```
<bean class="ares.remoting.framework.provider.ProviderFactoryBean" lazy-
init="false">

    <!-- 服务接口 -->

    <property name="serviceItf" value="com.ares.remoting.test.HelloService
"/>

    <!-- 服务实现类 -->

    <property name="serviceObject">

        <bean class="com.ares.remoting.test.HelloServiceImpl"/>

    </property>

    <!-- 应用标识，可根据实际需要配置不同的值 -->

    <property name="appKey" value="ares"/>

    <!-- 服务权重，可选配置，默认值为 1 -->

    <property name="weight" value="2"/>

    <!-- 服务分组组名，可选配置，默认值 default -->

    <property name="groupName" value="default"/>

    <!-- 服务工作者线程，可选配置，默认值 10 -->

    <property name="workerThreads" value="11"/>

    <!-- 服务端口 -->
```

```
<property name="serverPort" value="8081"/>
<!-- 服务超时时间 -->
<property name="timeout" value="600"/>
</bean>
```

4.4.2 实现远程服务的引入

远程服务的引入相关的属性定义如下：

```
//服务接口
private Class<?> targetInterface;
//超时时间
private int timeout;
//服务 bean
private Object serviceObject;
//负载均衡策略
private String clusterStrategy;
//服务提供者唯一标识
private String remoteAppKey;
//服务分组组名
private String groupName = "default";
```

引入远程服务需要做的操作如下。

- ◎ 通过注册中心，将服务提供者信息获取到本地缓存列表。
- ◎ 初始化 Netty 连接池。
- ◎ 获取服务提供者代理对象。
- ◎ 将服务消费者信息注册到注册中心。

具体实现代码如下。

```
1  import ares.remoting.framework.model.InvokerService;
2  import ares.remoting.framework.model.ProviderService;
3  import ares.remoting.framework.zookeeper.IRegisterCenter4Invoker;
4  import ares.remoting.framework.zookeeper.RegisterCenter;
5  import org.apache.commons.collections.MapUtils;
6  import org.springframework.beans.factory.FactoryBean;
7  import org.springframework.beans.factory.InitializingBean;
8
9  import java.util.List;
10 import java.util.Map;
11
12 public class RevokerFactoryBean implements FactoryBean,
InitializingBean {
13
14     //服务接口
15     private Class<?> targetInterface;
16     //超时时间
17     private int timeout;
18     //服务 bean
19     private Object serviceObject;
20     //负载均衡策略
21     private String clusterStrategy;
22     //服务提供者唯一标识
23     private String remoteAppKey;
24     //服务分组组名
25     private String groupName = "default";
26
27     @Override
28     public Object getObject() throws Exception {
29         return serviceObject;
30     }
31 }
```

```
32     @Override
33     public Class<?> getObjectType() {
34         return targetInterface;
35     }
36
37
38     @Override
39     public boolean isSingleton() {
40         return true;
41     }
42
43     @Override
44     public void afterPropertiesSet() throws Exception {
45
46         //获取服务注册中心
47         IRegisterCenter4Invoker registerCenter4Consumer =
48 RegisterCenter.singleton();
49         //初始化服务提供者列表到本地缓存
50         registerCenter4Consumer.initProviderMap(remoteAppKey,
groupNames);
51
52         //初始化 Netty Channel
53         Map<String, List<ProviderService>> providerMap =
54 registerCenter4Consumer.getServiceMetaDataMap4Consume();
55         if (MapUtils.isEmpty(providerMap)) {
56             throw new RuntimeException("service provider list is
empty.");
57         }
58
59         NettyChannelPoolFactory.channelPoolFactoryInstance().
60 initChannelPoolFactory(providerMap);
61
```



```
62         //获取服务提供者代理对象
63         RevokerProxyBeanFactory proxyFactory =
64         RevokerProxyBeanFactory.        singleton(targetInterface,        timeout,
clusterStrategy);
65         this.serviceObject = proxyFactory.getProxy();
66
67         //将消费者信息注册到注册中心
68         InvokerService invoker = new InvokerService();
69         invoker.setServiceItf(targetInterface);
70         invoker.setRemoteAppKey(remoteAppKey);
71         invoker.setGroupName(groupName);
72         registerCenter4Consumer.registerInvoker(invoker);
73     }
74
75
76     public Class<?> getTargetInterface() {
77         return targetInterface;
78     }
79
80     public void setTargetInterface(Class<?> targetInterface) {
81         this.targetInterface = targetInterface;
82     }
83
84     public int getTimeout() {
85         return timeout;
86     }
87
88     public void setTimeout(int timeout) {
89         this.timeout = timeout;
90     }
91
92     public Object getServiceObject() {
```

```
93         return serviceObject;
94     }
95
96     public void setServiceObject(Object serviceObject) {
97         this.serviceObject = serviceObject;
98     }
99
100    public String getClusterStrategy() {
101        return clusterStrategy;
102    }
103
104    public void setClusterStrategy(String clusterStrategy) {
105        this.clusterStrategy = clusterStrategy;
106    }
107
108    public String getRemoteAppKey() {
109        return remoteAppKey;
110    }
111
112    public void setRemoteAppKey(String remoteAppKey) {
113        this.remoteAppKey = remoteAppKey;
114    }
115
116    public String getGroupName() {
117        return groupName;
118    }
119
120    public void setGroupName(String groupName) {
121        this.groupName = groupName;
122    }
123 }
```

其中，第 15~26 行定义了引入远程服务相关的属性。

- ◎ 服务接口 `class (targetInterface)`: 用来匹配从服务注册中心获取到本地缓存的服务提供者，得到匹配服务接口的服务提供者列表，再依据软负载策略选取某一个服务提供者，发起调用。
- ◎ 超时时间 (`timeout`): 服务调用超时时间，超过所设置的时间之后，调用方不再等待服务方返回结果，直接返回给调用方。
- ◎ 服务 Bean (`serviceObject`): 远程服务生成的调用方本地代理对象，可以看作调用方 Stub。
- ◎ 负载均衡策略 (`clusterStrategy`): 用于配置服务调用方软负载策略，一般包括随机策略、加权随机策略、轮询策略、加权轮询策略、源地址 hash 策略等负载算法。关于负载均衡，在第 7 章会专门讲解。
- ◎ 服务提供者唯一标识 (`remoteAppKey`): 与服务发布配置的 `appKey` 保持一致。
- ◎ 服务分组组名 (`groupName`): 与服务发布配置的 `groupName` 保持一致，用于实现同一个服务分组功能。

第 45~76 行，方法 `afterPropertiesSet()` 中主要做了四件事情。第一，通过方法 `registerCenter4Consumer.initProviderMap()` 从服务注册中心获取服务提供者信息到本地缓存。第二，通过方法 `NettyChannelPoolFactory.channelPoolFactoryInstance().initChannelPoolFactory` 根据服务提供者信息初始化 Netty Channel 连接池，通过连接池可以做到 Channel 长连接复用，有利于提高服务调用性能。第三，通过方法 `RevokerProxyBeanFactory.singleton()` 获取 `RevokerProxyBeanFactory` 服务提供者代理对象。第四，通过方法 `registerCenter4Consumer.registerInvoker` 将服务调用者信息注册到 ZooKeeper 服务注册中心，为服务治理功能做数据准备。

其中远程服务代理类 `RevokerProxyBeanFactory` 的实现请参考本书第 7 章软负载实现的讲解。

下面是引入远程服务 `HelloService` 的 XML 配置：

```

<bean id="remoteHelloService"
      class="ares.remoting.framework.revoker.RevokerFactoryBean" lazy-
init="false">
    <!-- 远程服务接口 -->
    <property name="targetInterface" value="com.ares.remoting.test.H
elloService"/>
    <!-- 配置软负载策略，可选配置，默认为 Random 策略，其余配置项参
考 ClusterStrategyEnum -->
    <property name="clusterStrategy" value="WeightRandom"/>
    <!-- 远程服务应用标识 ares，可根据实际需要配置不同的值-->
    <property name="remoteAppKey" value="ares"/>
    <!-- 服务分组组名，可选配置，默认为 default -->
    <property name="groupName" value="default"/>
    <!-- 客户端超时时间 -->
    <property name="timeout" value="600"/>
</bean>

```

4.5 在 Spring 中定制自己的 XML 标签

上节发布与引入服务的 XML 配置有些烦琐，为了简化服务的发布与引入，可以定制独有的 Spring 标签，使得我们的远程服务发布与引入更加便捷。在本节，将介绍如何在 Spring 中定制自己的 XML 标签。

先定义自定义标签的命名空间，路径与构成规则，首先在工程目录 `src/main/resources/META-INF` 下定义文件 `remote-reference.xsd`，文件内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.ares-remoting.com/schema/ares-reference"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"

```

```

xmlns:beans="http://www.springframework.org/schema/beans"

targetNamespace="http://www.ares-remoting.com/schema/ares-reference"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

<xsd:import namespace="http://www.springframework.org/schema/beans"/>

<xsd:element name="reference">
<xsd:complexType>
<xsd:complexContent>
<xsd:extension base="beans:identifiedType">
<xsd:attribute name="interface" type="xsd:string"/>
<xsd:attribute name="timeout" type="xsd:int" use="required"/>
<xsd:attribute name="clusterStrategy" type="xsd:string" use="optional"/>
<xsd:attribute name="remoteAppKey" type="xsd:string" use="required"/>
<xsd:attribute name="groupName" type="xsd:string" use="optional"/>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

remote-service.xsd 文件内容如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.ares-remoting.com/schema/ares-service"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:beans="http://www.springframework.org/schema/beans"
    targetNamespace="http://www.ares-remoting.com/schema/
ares-service"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

```

```
<xsd:import namespace="http://www.springframework.org/schema/beans"/>

<xsd:element name="service">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="beans:identifiedType">
        <xsd:attribute name="interface" type="xsd:string" use="required"/>
        <xsd:attribute name="timeout" type="xsd:int" use="required"/>
        <xsd:attribute name="serverPort" type="xsd:int" use="required"/>
        <xsd:attribute name="ref" type="xsd:string" use="required"/>
        <xsd:attribute name="weight" type="xsd:int" use="optional"/>
        <xsd:attribute name="workerThreads" type="xsd:int" use="optional"/>
        <xsd:attribute name="appKey" type="xsd:string" use="required"/>
        <xsd:attribute name="groupName" type="xsd:string" use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

spring.schemas 文件内容如下：

```
http://www.ares-remoting.com/schema/ares-service.xsd=META-INF/remote-service.xsd
```

```
http://www.ares-remoting.com/schema/ares-reference.xsd=META-INF/remote-reference.xsd
```

spring.handlers 文件内容如下：

```
http://www.ares-remoting.com/schema/ares-service=ares.remoting.framework.spring.AresRemoteServiceNamespaceHandler
```

```
http://www.ares-remoting.com/schema/ares-reference=ares.remoting.framework.spring.AresRemoteReferenceNamespaceHandler
```

定义解析自定义标签的工具类 `AresRemoteReferenceNamespaceHandler`，为远程服务引入自定义标签处理类：

```
import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class AresRemoteReferenceNamespaceHandler extends
NamespaceHandlerSupport {
    @Override
    public void init() {
        registerBeanDefinitionParser("reference", new
RevokerFactoryBeanDefinitionParser());
    }
}
```

`RevokerFactoryBeanDefinitionParser` 解析远程服务引入的自定义标签类，解析标签中配置的 XML 属性值，将属性值写入远程服务引入的 Bean 中，完成服务引入对象的构建：

```
import ares.remoting.framework.revoker.RevokerFactoryBean;
import org.apache.commons.lang.StringUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import
org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.w3c.dom.Element;

public class RevokerFactoryBeanDefinitionParser extends AbstractSingleBean
DefinitionParser {

    //logger
    private static final Logger logger = LoggerFactory.getLogger
(RevokerFactoryBeanDefinitionParser.class);

    protected Class getBeanClass(Element element) {
```

```
        return RevokerFactoryBean.class;
    }

    protected void doParse(Element element, BeanDefinitionBuilder bean) {

        try {
            String timeOut = element.getAttribute("timeout");
            String targetInterface = element.getAttribute("interface");
            String clusterStrategy = element.getAttribute("clusterStrategy");
            String remoteAppKey = element.getAttribute("remoteAppKey");
            String groupName = element.getAttribute("groupName");

            bean.addPropertyValue("timeout", Integer.parseInt(timeOut));
            bean.addPropertyValue("targetInterface",
Class.forName(targetInterface));
            bean.addPropertyValue("remoteAppKey", remoteAppKey);

            if (StringUtils.isNotBlank(clusterStrategy)) {
                bean.addPropertyValue("clusterStrategy", clusterStrategy);
            }

            if (StringUtils.isNotBlank(groupName)) {
                bean.addPropertyValue("groupName", groupName);
            }
        } catch (Exception e) {
            logger.error("RevokerFactoryBeanDefinitionParser error.", e);
            throw new RuntimeException(e);
        }
    }
}
```

AresRemoteServiceNamespaceHandler 为远程服务发布自定义标签处理类：

```
import org.springframework.beans.factory.xml.NamespaceHandlerSupport;
```



```

public class AresRemoteServiceNamespaceHandler extends NamespaceHandler
Support {
    @Override
    public void init() {
        registerBeanDefinitionParser("service", new
ProviderFactoryBeanDefinitionParser());
    }
}

```

ProviderFactoryBeanDefinitionParser 的作用是解析服务发布自定义标签，获得 XML 标签配置的属性值，完成服务发布对象的构建：

```

import ares.remoting.framework.provider.ProviderFactoryBean;
import org.apache.commons.lang.StringUtils;
import org.apache.commons.lang.math.NumberUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBean
DefinitionParser;
import org.w3c.dom.Element;

public class ProviderFactoryBeanDefinitionParser extends AbstractSingle
BeanDefinitionParser {

    //logger
    private static final Logger logger = LoggerFactory.getLogger
(ProviderFactoryBeanDefinitionParser.class);

    protected Class getBeanClass(Element element) {
        return ProviderFactoryBean.class;
    }
}

```

```
protected void doParse(Element element, BeanDefinitionBuilder bean) {

    try {
        String serviceItf = element.getAttribute("interface");
        String timeOut = element.getAttribute("timeout");
        String serverPort = element.getAttribute("serverPort");
        String ref = element.getAttribute("ref");
        String weight = element.getAttribute("weight");
        String workerThreads = element.getAttribute("workerThreads");
        String appKey = element.getAttribute("appKey");
        String groupName = element.getAttribute("groupName");

        bean.addPropertyValue("serverPort",
Integer.parseInt(serverPort));
        bean.addPropertyValue("timeout", Integer.parseInt(timeOut));
        bean.addPropertyValue("serviceItf",
Class.forName(serviceItf));
        bean.addPropertyReference("serviceObject", ref);
        bean.addPropertyValue("appKey", appKey);

        if (NumberUtils.isNumber(weight)) {
            bean.addPropertyValue("weight", Integer.parseInt(weight));
        }
        if (NumberUtils.isNumber(workerThreads)) {
            bean.addPropertyValue("workerThreads",
Integer.parseInt(workerThreads));
        }
        if (StringUtils.isNotBlank(groupName)) {
            bean.addPropertyValue("groupName", groupName);
        }
    } catch (Exception e) {
```

```

        logger.error("ProviderFactoryBeanDefinitionParser error.", e);
        throw new RuntimeException(e);
    }
}
}

```

最后，基于以上实现，使用自定义 Spring 标签发布服务的配置如下：

```

<!-- 发布远程服务 -->
<bean id="helloService" class="com.ares.remoting.test.HelloServiceImpl
"/>

<AresServer:service id="helloServiceRegister"
    interface="com.ares.remoting.test.HelloService"
    ref="helloService"
    groupName="default"
    weight="2"
    appKey="ares"
    workerThreads="11"
    serverPort="8081"
    timeout="600"/>

```

使用自定义 Spring 标签引入服务的配置如下：

```

<!-- 引入远程服务 -->
<AresClient:reference id="remoteHelloService"
    interface="com.ares.remoting.test.HelloService"
    clusterStrategy="WeightRandom"
    remoteAppKey="ares"
    groupName="default"
    timeout="600"/>

```

以上配置实现服务发布与引入效果与 4.4 节中的服务发布与引入配置是一样的。

4.6 本章小结

本章对 Spring 整体架构做了概要性介绍。重点讲解了 `FactoryBean` 的特性及如何利用 `FactoryBean` 来完成远程服务的发布与引入。同时详细介绍了 Spring 如何使用 `FactoryBean` 来完成对 RMI、HTTP Invoker 及 Hessian 等 RPC 框架的集成，并给出了实际可运行的集成示例，有助于读者进一步理解 `FactoryBean` 的应用。最后，讲解了如何将本次开发的分布式服务框架使用 `FactoryBean` 实现服务的发布与引入功能，完成与 Spring 的无缝集成。同时，为了简化服务发布与引入的配置，介绍了如何定制开发 Spring 标签。

第 5 章

分布式服务框架注册中心

5.1 服务注册中心介绍

分布式服务框架部署在多台不同的机器上，例如服务提供者部署在集群 A，服务调用者部署在集群 B，在服务调用的过程中，集群 A 中的机器需要与集群 B 中的机器进行通信，如图 5-1 所示。

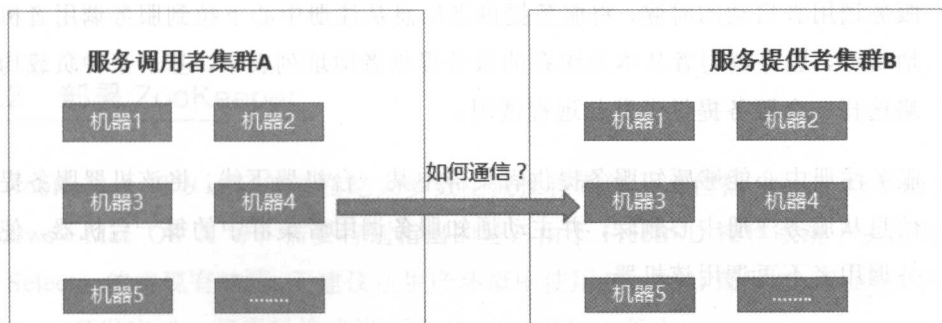


图 5-1 服务提供者集群与服务调用者集群

有如下问题需要解决。

- ◎ 集群 A 中的服务调用者如何发现集群 B 中的服务提供者。
- ◎ 集群 A 中的服务调用者如何选择集群 B 中的某一台服务提供者机器发起调用。
- ◎ 集群 B 中的服务某台提供者机器下线之后，集群 A 中的服务调用者如何感知到这台机器的下线，不再对已下线的机器发起调用。
- ◎ 集群 B 提供的某个服务如何获知集群 A 中哪些机器正在消费该服务。

以上问题将通过服务注册中心来解决，我们采用服务注册中心来实时存储更新服务提供者信息及该服务的实时调用者信息，如图 5-2 所示。

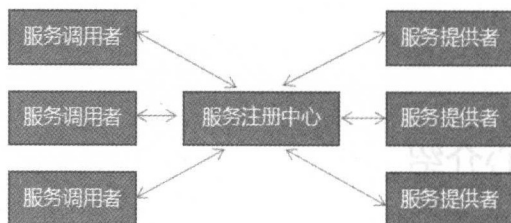


图 5-2 服务调用者、服务注册中心、服务提供者关系示意图

- ◎ 在服务启动的时候，将服务提供者信息主动上报到服务注册中心进行服务注册。
- ◎ 服务调用者启动的时候，将服务提供者信息从注册中心下拉到服务调用者机器本地缓存，服务调用者从本地缓存的服务提供者地址列表中，基于某种负载均衡策略选择一台服务提供者发起远程调用。
- ◎ 服务注册中心能够感知服务提供者集群中某一台机器下线，将该机器服务提供者信息从服务注册中心删除，并主动通知服务调用者集群中的每一台机器，使得服务调用者不再调用该机器。

服务注册中心有以下优点。

- ◎ 软负载及透明化服务路由：服务提供者和服务调用者之间互相解耦，服务调用者不需要硬编码服务提供者地址。
- ◎ 服务动态发现及可伸缩能力：服务提供者机器增减能被服务调用者通过注册中心动态感知，而且通过增减机器可以实现服务的弹性伸缩。
- ◎ 通过注册中心可以动态地监控服务运行质量及服务依赖，为服务提供服务治理能力。

5.2 ZooKeeper 实现服务的注册中心原理

5.2.1 ZooKeeper 介绍

Apache ZooKeeper (<http://zookeeper.apache.org/>) 是由雅虎开发并开源的分布式协调服务，现已成为 Apache 的顶级项目之一。ZooKeeper 提供了统一命名服务、配置管理、分布式锁等分布式基础服务，基于这些基础服务，我们可以实现集群管理、软负载、发布/订阅、分布式锁、分布式队列、命名服务等功能。

ZooKeeper 已经被广泛用于分布式系统的开发，像 Hbase、Kafka、Solr 等知名开源项目均使用了 ZooKeeper 来实现其某种分布式基础服务。

5.2.2 部署 ZooKeeper

ZooKeeper 可以运行部署在绝大多数主流操作系统上，包括 Linux 众多的发行版、Windows、Mac OS X 等。需要特别指出的是，由于 FreeBSD 操作系统下实现的 JVM 中 NIO Selector 的实现有缺陷，不建议在生产环境中使用 FreeBSD 部署 ZooKeeper。ZooKeeper 使用 Java 开发完成，部署环境建议安装 JDK1.6 及以上版本。

ZooKeeper 有三种部署模式：单机模式、集群模式、伪集群模式。

1. 单机模式

单机模式就是将 ZooKeeper 部署在一台机器上，使用单独的一台机器来提供 ZooKeeper 服务。单机部署模式部署简单，一般用在开发或者测试环境。Linux 操作系统下安装单机模式，具体步骤如下。

(1) 安装 JDK1.6 或 JDK1.6 以上版本的 Java 运行时环境。

(2) 通过 <http://zookeeper.apache.org/releases.html> 下载安装包，例如 zookeeper-3.4.8.tar.gz，解压到合适的目录，例如/Users/liyebing/open_source_pro/zookeeper/zookeeper-3.4.8 目录下，目录结构如图 5-3 所示。

CHANGES.txt	bin	docs	src
LICENSE.txt	build.xml	ivy.xml	zookeeper-3.4.8.jar
NOTICE.txt	conf	ivysettings.xml	zookeeper-3.4.8.jar.asc
README.txt	contrib	lib	zookeeper-3.4.8.jar.md5
README_packaging.txt	dist-maven	recipes	zookeeper-3.4.8.jar.sha1

图 5-3 ZooKeeper 解压后的目录结构

(3) 进入 conf 目录，创建 zoo.cfg 文件，也可以将已经存在的文件 zoo_sample.cfg 重命名为 zoo.cfg，内容配置如下（代码注释中 ZooKeeper 简写为 ZK）：

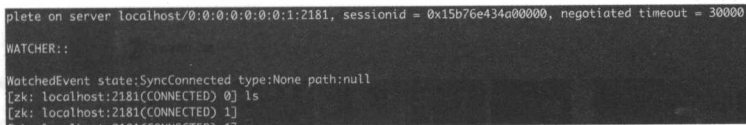
```
tickTime=2000      #单位毫秒，ZK 中最小时间单位长度
dataDir=/Users/liyebing/zookeeper/data      #ZK 服务器存储快照文件目录
dataLogDir=/Users/liyebing/zookeeper/logs #ZK 服务器存储事务日志文件目录
clientPort=2181     #服务端对外的服务端口
initLimit=5         #tickTime 的倍数，Leader 等待 Follower 启动并完成数据同步的时间
syncLimit=2         #tickTime 的倍数，Leader 与 Follower 之间心跳最大延时时间
server.1=127.0.0.1:2888:3888 #ZK 服务地址，server.1 中的数字 1 标识集群中机器序号
```

(4) 启动服务，进入 ZooKeeper 解压后的子目录/bin 目录下，运行 sh zkServer.sh start，如图 5-4 所示。

```
yb:bin liyebing$ sh zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /Users/liyebing/open_source_pro/zookeeper/zookeeper-3.4.8/bin/../conf/zoo.cfg
-n Starting zookeeper ...
STARTED
```

图 5-4 运行 ZooKeeper 服务端

(5) 验证服务是否启动正常。同样在/bin目录下,运行 ZooKeeper 的命令行客户端 sh zkCli.sh。若能正常连接上服务端,则说明服务启动正常,如图 5-5 所示。



```

connect to server localhost/0:0:0:0:0:0:0:1:2181, sessionId = 0x15b76e43a00000, negotiated timeout = 30000
WATCHER::
WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls
[zk: localhost:2181(CONNECTED) 1]

```

图 5-5 使用 ZooKeeper 命令行客户端连接 ZooKeeper 服务

2. 集群模式

比如我们使用三台 Linux 机器来构建 ZooKeeper 集群,具体步骤如下。

(1) 分别在三台机器安装 JDK1.6 或 JDK1.6 以上版本的 Java 运行时环境及下载 ZooKeeper 安装包。

(2) 配置 zoo.cfg 文件。

内容配置如下:

```

tickTime=2000      #单位毫秒, ZK 中最小时间单位长度
dataDir=/Users/liyebing/zookeeper/data      #ZK 服务器存储快照文件目录
dataLogDir=/Users/liyebing/zookeeper/logs    #ZK 服务器存储事务日志文件目录
clientPort=2181     #服务端对外的服务端口
initLimit=5         #tickTime 的倍数, Leader 等待 Follower 启动并完成数据同步
                    的时间
syncLimit=2         #tickTime 的倍数, Leader 与 Follower 之间心跳最大延时时间
server.1=IPA:2888:3888 #ZK 服务地址, server.1 中的数字 1 标识集群中机器序号
server.2=IPB:2888:3888 #ZK 服务地址, server.2 中的数字 2 标识集群中机器序号
server.3=IPC:2888:3888 #ZK 服务地址, server.3 中的数字 3 标识集群中机器序号

```

其中 IPA、IPB、IPC 对应集群中三台 Linux 机器的 IP。

(3) 配置 myid 文件

在 dataDir 文件夹下,创建一个 myid 文件,文件第 1 行写上与 zoo.cfg 中 server.id=host:

port:port 当前机器对应的 server.id 配置中的机器编号。比如，机器 IPA，myid 的内容为 1。

3. 伪集群模式

伪集群模式实际是使用一台机器以集群的模式对外提供 ZooKeeper 服务。伪集群模式与集群模式的不同点在于 zoo.cfg 的配置：

```

tickTime=2000      #单位毫秒，ZK 中最小时间单位长度
dataDir=/Users/liyebing/zookeeper/data      #ZK 服务器存储快照文件目录
dataLogDir=/Users/liyebing/zookeeper/logs    #ZK 服务器存储事务日志文件目录
clientPort=2181     #服务端对外的服务端口
initLimit=5         #tickTime 的倍数，Leader 等待 Follower 启动并完成数据同步的时间
syncLimit=2         #tickTime 的倍数，Leader 与 Follower 之间心跳最大延时时间
server.1=IPA:2888:3888
server.2=IPA:2889:3889
server.3=IPA:2890:3890

```

其中，server.id 后面对应的 IP 地址都配置为当前机器的 IP，但因为需要在一台机器上启动多个进程，故后面的端口配置不一样。

5.2.3 ZkClient 使用介绍

ZooKeeper 原生的 Java API 使用起来不太方便，ZkClient 是基于 ZooKeeper 原生的 Java API 开发的一个易用性更好的客户端，实现了 Session 超时自动重连、Watcher 反复注册等功能，减轻了开发人员的负担。

5.2.3.1 ZkClient 主要 API 介绍

创建会话连接主要 API：

```

/**
 * zkServers 为格式为 host1:port1,host2:port2 组成的字符串
 * connectionTimeout 创建连接的超时时间，单位为 ms
 */

```

```

public ZkClient(String zkServers, int connectionTimeout)

/**
 * zkServers 为格式为 host1:port1,host2:port2 组成的字符串
 * sessionTimeout 会话超时时间, 单位为 ms
 * connectionTimeout 创建连接的超时时间, 单位为 ms
 */
public ZkClient(String zkServers, int sessionTimeout, int
connectionTimeout)

/**
 * zkServers 为格式为 host1:port1,host2:port2 组成的字符串
 * sessionTimeout 会话超时时间, 单位为 ms
 * connectionTimeout 创建连接的超时时间, 单位为 ms
 * zkSerializer 自定义 ZK 节点存储数据的序列化方式
 */
public ZkClient(String zkServers, int sessionTimeout, int
connectionTimeout, ZkSerializer zkSerializer)

/**
 * connection IZkConnection 接口自定义实现
 */
public ZkClient(IZkConnection connection)

/**
 * connection IZkConnection 接口自定义实现
 * connectionTimeout 创建连接的超时时间, 单位为 ms
 */
public ZkClient(IZkConnection connection, int connectionTimeout)

/**
 * connection IZkConnection 接口自定义实现
 * connectionTimeout 创建连接的超时时间, 单位为 ms

```

```
    * zkSerializer 自定义 ZK 节点存储数据的序列化方式
    */
    public ZkClient(IZkConnection connection, int connectionTimeout,
        ZkSerializer zkSerializer)
```

对于 IZkConnection, ZkClient 默认提供了两种实现, 分别是 ZKConnection 与 InMemoryConnection。一般默认使用 ZKConnection 即可满足绝大多数使用场景的需要。

对于 ZkSerializer, ZkClient 默认使用 Java 自带的序列化方式, 当然也可以自己实现 ZkSerializer 序列化器。

创建节点主要 API:

```
/**
 * 创建临时节点, 同时写入 data 数据
 *
 * @param path
 * @param data
 */
public void createEphemeral(final String path, final Object data);

/**
 * 创建有序临时节点, 同时写入 data 数据
 * @param path
 * @param data
 * @return
 */
public String createEphemeralSequential(final String path, final Object
data);

/**
 * 创建临时节点
 *
 * @param path
```

```
*/
public void createEphemeral(final String path);

/**
 * 创建有序持久节点，同时写入 data 数据
 *
 * @param path
 * @param data
 * @return
 */
public String createPersistentSequential(String path, Object data);

/**
 * 创建持久节点，同时写入 data 数据
 *
 * @param path
 * @param data
 */
public void createPersistent(String path, Object data);

/**
 * 创建持久节点，若父节点不存在，可自动创建父节点
 *
 * @param path
 * @param createParents
 */
public void createPersistent(String path, boolean createParents);

/**
 * 创建持久节点
 *
 * @param path
 */
*/
```

```
public void createPersistent(String path);
```

从方法名称基本上就能知道每个方法的作用。其中，原生的 ZooKeeper Java API 是无法递归创立节点的，只能逐级创建节点，ZkClient 通过 createParents 参数递归逐级创建节点，极大地方便了开发人员。

删除节点主要 API:

```
/**
 * 删除指定节点
 *
 * @param path
 * @return
 */
public boolean delete(final String path);
```

```
/**
 * 递归删除 path 路径的所有节点
 *
 * @param path
 * @return
 */
public boolean deleteRecursive(String path);
```

在 ZooKeeper 中，只允许逐级删除叶子节点，如果要删除一个父节点，必须先逐一删除所有子节点。deleteRecursive 方法自动逐级删除所有子节点，为开发人员提供了很大的便利。

读写节点数据主要 API 如下:

```
/**
 * 获取指定节点的子节点列表
 *
```

```

    * @param path
    * @return
    */
    public List<String> getChildren(String path);

    /**
     * 注册监听器 listener, 监听路径 path 下子节点的变化
     *
     * @param path
     * @param listener
     * @return
     */
    public List<String> subscribeChildChanges(String path, IZkChildListener
listener);

    /**
     * 注册监听器 listener, 监听路径 path 下子节点数据的变化
     *
     * @param path
     * @param listener
     */
    public void subscribeDataChanges(String path, IZkDataListener
listener);

    /**
     * 读取数据
     *
     * @param path
     * @param <T>
     * @return
     */
    public <T extends Object> T readData(String path);

```

```
/**
 * 读取数据，若节点不存在，则返回 null
 *
 * @param path
 * @param returnNullIfPathNotExists
 * @param <T>
 * @return
 */
public <T extends Object> T readData(String path, boolean
returnNullIfPathNotExists);

/**
 * 指定节点的状态信息，读取节点数据
 *
 * @param path
 * @param stat
 * @param <T>
 * @return
 */
public <T extends Object> T readData(String path, Stat stat);

/**
 * 写入或者更新预期数据版本的节点数据
 *
 * @param path
 * @param datat
 * @param expectedVersion
 */
public void writeData(final String path, Object datat, final int
expectedVersion);

/**
```



```

    * 写入或者更新数据
    *
    * @param path
    * @param object
    */
    public void writeData(String path, Object object);

```

5.2.3.2 ZkClient 使用示例

下面我们将通过 ZkClient 来实现诸如创建节点, 对节点数据的增删改查等操作来演示 ZkClient API 的使用。

先引入 ZkClient 的 Maven 依赖:

```

<!--zookeeper 依赖 -->
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.8</version>
</dependency>
<!--zkclient 依赖 -->
<dependency>
    <groupId>com.github.sgroschupf</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.1</version>
</dependency>

```

演示 ZkClient API 的使用代码如下。

```

public class ZKClientDemo {

    public static void main(String[] args) throws Exception {
        //创建会话
        String zkServers = "127.0.0.1:2181";
        int connectionTimeout = 3000;
    }
}

```

```
ZkClient zkClient = new ZkClient(zkServers, connectionTimeout);

String path = "/zk-data";
//若节点已经存在，则删除
if (zkClient.exists(path)) {
    zkClient.delete(path);
}
//创建持久节点
zkClient.createPersistent(path);

//节点写入数据
zkClient.writeData(path, "test_data_1");

//节点读取数据
String data = zkClient.<String>readData(path, true);
System.out.println(data);

//注册监听器，监听数据变化
zkClient.subscribeDataChanges(path, new IZkDataListener() {
    public void handleDataChange(String dataPath, Object data) throws
Exception {
        System.out.println("handleDataChange,dataPath:" + dataPath +
" data:" + data);
    }

    public void handleDataDeleted(String dataPath) throws Exception {
        System.out.println("handleDataDeleted,dataPath:" + dataPath);
    }
});

//修改数据
zkClient.writeData(path, "test_data_2");
```

```

//删除节点
zkClient.delete(path);

Thread.sleep(Integer.MAX_VALUE);
}
}

```

以上代码，先连接 ZooKeeper 服务端，创建会话，然后通过 exists()检测所需要创建的节点是否存在，若存在，则调用 delete()删除。然后调用 createPersistent()创建持久节点。接着通过 API writeData()向该节点写入字符串数据"test_data_1"，紧接着通过 readData 方法读取写入的数据，并打印到控制台。再通过 subscribeDataChanges()注册该路径的数据变化监听器，如果数据被修改或者删除，会回调 IZkDataListener 接口中对应的方法。最后通过 writeData()修改节点下的数据为"test_data_2"，再使用 delete()方法删除节点。

运行以上代码，运行结果如下：

```

test_data_1
handleDataDeleted,dataPath:/zk-data
handleDataDeleted,dataPath:/zk-data

```

可以看到，通过 writeData()修改数据或者通过 delete()删除数据的时候，都回调了 IZkDataListener 对应的接口方法。

5.2.4 ZooKeeper 实现服务注册中心

在基于 SOA 架构的应用中，应用提供对外服务的同时也会调用外部系统提供的服务。当应用越来越多，服务越来越多，服务之间的依赖越来越复杂，这个时候依靠人工来管理服务之间的依赖以服务的上下线已经变得不可能。这种情况下，我们需要服务注册中心来解决服务自动动态发现、服务自动上下线等问题。下面介绍如何使用 ZooKeeper 来实现服务注册中心。

服务的服务端服务启动的同时，将服务提供者信息（主机 IP 地址、服务端口、服务

接口类路径)组成的 `znode` 路径写入 `ZooKeeper` 中, 注意写入的叶子节点为临时节点。这样就完成了服务的注册动作。

服务的消费端在发起服务调用之前, 会先连接到 `ZooKeeper`, 对服务提供者节点路径注册监听器, 同时获取服务提供者信息到本地缓存, 发起调用的时候, 调用者会从服务提供者本地缓存列表中运用某种负载均衡策略选取某一个服务提供者, 对该服务提供者发起调用, 最终完成本次服务调用。这样就完成了服务发现动作。

若服务提供者集群中某台机器下线, 该机器与注册中心 `ZooKeeper` 的连接会断掉, 因为服务注册写入信息的叶子节点写入的 `znode` 是临时节点。故当与 `ZooKeeper` 连接断掉后, 该临时节点会被自动删除。同时触发服务消费端对服务提供者路径的监听器, 服务消费端收到被删除服务提供者节点信息之后, 刷新本地服务提供者信息缓存, 从缓存中删除已下线的服务提供者信息。这样就做到了服务的自动下线。同理, 服务提供者集群新增机器, 会在服务提供者 `znode` 下新增临时叶子节点, 同时触发服务消费端对服务提供者路径的监听器, 服务消费端收到新增服务提供者节点信息之后, 刷新本地服务提供者信息缓存, 将新加入的服务提供者信息加入服务提供者信息本地缓存中。这样就做到了服务的自动扩容上线。

此外, 我们还可以通过服务注册中心来收集服务消费者信息, 以达到实现部分服务治理功能的目的。在服务消费端, 通过将服务消费者信息写入 `ZooKeeper` 临时节点, 一旦消费者机器下线, 断开与 `ZooKeeper` 的连接, 该临时节点将被自动删除, 达到通过 `ZooKeeper` 自动收集消费者信息的目的。

图 5-6 给出了 `ZooKeeper` 实现注册中心的节点树结构。第 1 层节点 `APPKEY`, 用来唯一标识一个应用, 可以看作该应用的一个命名空间。第 2 层节点 `Service` 用来存储实现服务的类信息, 第 3 层节点用来区分服务消费者与服务提供者, 最后一层节点存储服务消费者或者提供者的主机 IP 与服务端口。

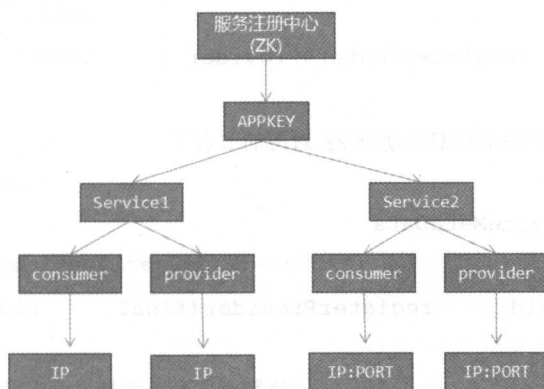


图 5-6 服务注册中心 ZK 节点树示意图

5.3 集成 ZooKeeper 实现自己的服务注册与发现

本节我们将使用 ZooKeeper 的相关特性实现一个服务注册中心。服务注册中心有两个使用方，分别是服务提供方与服务消费方。所以，我们实现两个接口，来给出服务提供方与消费方各自的方法定义。

5.3.1 服务注册中心服务提供方

注册中心服务提供方接口方法主要有：

- ◎ 服务注册方法 `registerProvider()`。
- ◎ 获取所有服务提供者信息方法 `getProviderServiceMap()`。

```
import java.util.List;
```

```
import java.util.Map;
```

```
/**
```

```
* 服务端注册中心接口
```

```
*/
public interface IRegisterCenter4Provider {
    /**
     * 服务端将服务提供者信息注册到 zk 对应的节点下
     *
     * @param serviceMetaData
     */
    public void registerProvider(final List<ProviderService>
serviceMetaData);

    /**
     * 服务端获取服务提供者信息
     * <p/>
     * 注：返回对象，Key：服务提供者接口 value：服务提供者服务方法列表
     *
     * @return
     */
    public Map<String, List<ProviderService>> getProviderServiceMap();
}
```

5.3.2 服务注册中心服务消费方

服务注册中心服务消费方的方法主要有：

- ◎ 初始化服务提供者信息本地缓存 `initProviderMap()`。
- ◎ 消费端获取服务提供者信息 `getServiceMetaDataMap4Consume()`。
- ◎ 消费端将消费者信息注册方法 `registerInvoker()`。

```
import ares.remoting.framework.model.InvokerService;
import ares.remoting.framework.model.ProviderService;
```

```
import java.util.List;
import java.util.Map;

/**
 * 消费端注册中心接口
 */
public interface IRegisterCenter4Invoker {

    /**
     * 消费端初始化服务提供者信息本地缓存
     */
    public void initProviderMap();

    /**
     * 消费端获取服务提供者信息
     *
     * @return
     */
    public Map<String, List<ProviderService>>getServiceMetaDataMap4
Consume();

    /**
     * 消费端将消费者信息注册到 ZK 对应的节点下
     *
     * @param invoker
     */
    public void registerInvoker(final InvokerService invoker);
}
```

5.3.3 服务注册中心实现

下面给出注册中心的实现主体代码。

```

1  import ares.remoting.framework.Helper.IPHelper;
2  import ares.remoting.framework.Helper.PropertyConfigHelper;
3  import ares.remoting.framework.model.InvokerService;
4  import ares.remoting.framework.model.ProviderService;
5  import com.google.common.collect.Maps;
6  import com.alibaba.fastjson.JSON;
7  import com.google.common.base.Function;
8  import com.google.common.collect.Lists;
9  import org.I0Itec.zkclient.IZkChildListener;
10 import org.I0Itec.zkclient.ZkClient;
11 import org.I0Itec.zkclient.serialize.SerializableSerializer;
12 import org.apache.commons.collections.CollectionUtils;
13 import org.apache.commons.collections.MapUtils;
14 import org.apache.commons.lang3.ClassUtils;
15 import org.apache.commons.lang3.StringUtils;
16
17 import java.util.List;
18 import java.util.Map;
19
20 /**
21  * 注册中心实现
22  */
23 public class RegisterCenter implements IRegisterCenter4Invoker,
24 IRegisterCenter4Provider {
25
26     private static RegisterCenter registerCenter = new RegisterCenter();
27
28     //服务提供者列表, Key: 服务提供者接口 value: 服务提供者服务方法列表
29     private static final Map<String, List<ProviderService>>

```



```

30 providerServiceMap = Maps.newConcurrentMap();
31 //服务端 ZK 服务元信息, 选择服务 (第一次直接从 ZK 拉取, 后续由 ZK 的监听机制主动
更新)
32 private static final Map<String, List<ProviderService>>
33 serviceMetaDataMap4Consume =
com.google.common.collect.Maps.newConcurrentMap();
34 //从配置文件中获取 ZK 的服务地址列表
35 private static String ZK_SERVICE =
PropertyConfigHelper.getZkService();
36 //从配置文件中获取 ZK 会话超时时间配置
37 private static int ZK_SESSION_TIME_OUT =
PropertyConfigHelper.getZkConnectionTimeout();
38 //从配置文件中获取 ZK 连接超时时间配置
39 private static int ZK_CONNECTION_TIME_OUT =
PropertyConfigHelper.getZkConnectionTimeout();
40 //组装 ZK 根路径/APPKEY 路径
41 private static String ROOT_PATH = "/config_register/" +
PropertyConfigHelper.getAppName();
42 public static String PROVIDER_TYPE = "/provider";
43 public static String INVOKER_TYPE = "/consumer";
44 private static volatile ZkClient zkClient = null;
45
46 private RegisterCenter() {
47 }
48
49 public static RegisterCenter singleton() {
50 return registerCenter;
51 }
52
53 @Override
54 public void registerProvider(final List<ProviderService>
serviceMetaData) {
55 if (CollectionUtils.isEmpty(serviceMetaData)) {

```

```

59         return;
60     }
61
62     //连接 ZK, 注册服务
63     synchronized (RegisterCenter.class) {
64
65         for (ProviderService provider : serviceMetaData) {
66             String serviceItfKey =
provider.getServiceItf().getName();
67
68             List<ProviderService> providers = providerServiceMap.get
69 (serviceItfKey);
70             if (providers == null) {
71                 providers = Lists.newArrayList();
72             }
73             providers.add(provider);
74             providerServiceMap.put(serviceItfKey, providers);
75         }
76
77         if (zkClient == null) {
78             zkClient = new ZkClient(ZK_SERVICE, ZK_SESSION_TIME_OUT,
79 ZK_CONNECTION_TIME_OUT, new SerializableSerializer());
80         }
81         //创建 ZK 命名空间/当前部署应用 APP 命名空间
82         boolean exist = zkClient.exists(ROOT_PATH);
83         if (!exist) {
84             zkClient.createPersistent(ROOT_PATH, true);
85         }
86         //创建服务提供者节点
87         exist = zkClient.exists((ROOT_PATH));
88         if (!exist) {
89             zkClient.createPersistent(ROOT_PATH);
90         }

```

```

91
92         for (Map.Entry<String, List<ProviderService>> entry :
93 providerServiceMap.entrySet()) {
94             //创建服务提供者节点
95             String serviceNode = entry.getKey();
96             String servicePath = ROOT_PATH + "/" + serviceNode +
PROVIDER_TYPE;
97             exist = zkClient.exists(servicePath);
98             if (!exist) {
99                 zkClient.createPersistent(servicePath, true);
100             }
101
102             //创建当前服务器节点
103             int serverPort = entry.getValue().get(0).getServerPort();
104             String localIp = IPHelper.localIp();
105             String currentServiceIpNode = servicePath + "/" + localIp
106 + "|" + serverPort;
107             exist = zkClient.exists(currentServiceIpNode);
108             if (!exist) {
109                 //注意，这里创建的是临时节点
110                 zkClient.createEphemeral(currentServiceIpNode);
111             }
112
113             //监听注册服务的变化，同时更新数据到本地缓存
114             zkClient.subscribeChildChanges(servicePath, new
IZkChildListener() {
115                 @Override
116                 public void handleChildChange(String parentPath,
117 List<String> currentChilds) throws Exception {
118                     if (currentChilds == null) {
119                         currentChilds = Lists.newArrayList();
120                     }
121

```

```
122             //存活的服务 IP 列表
123             List<String> activityServiceIpList =
124             Lists.newArrayList(Lists.transform(currentChilds, new Function<String,
String>() {
125                 @Override
126                 public String apply(String input) {
127                     return StringUtils.split(input, "|")[0];
128                 }
129             }));
130             refreshActivityService(activityServiceIpList);
131         }
132     });
133
134     }
135 }
136 }
137
138 @Override
139 public Map<String, List<ProviderService>> getProviderServiceMap() {
140     return providerServiceMap;
141 }
142
143
144
145 @Override
146 public void initProviderMap() {
147     if (MapUtils.isEmpty(serviceMetaDataMap4Consume)) {
148         serviceMetaDataMap4Consume.putAll(fetchOrUpdateService
MetaData());
149     }
150 }
151
152 @Override
```

```

153     public Map<String, List<ProviderService>> getServiceMetaDataMap4
Consume() {
154         return serviceMetaDataMap4Consume;
155     }
156
157     @Override
158     public void registerInvoker(InvokerService invoker) {
159         if (invoker == null) {
160             return;
161         }
162
163         //连接 ZK, 注册服务
164         synchronized (RegisterCenter.class) {
165
166             if (zkClient == null) {
167                 zkClient = new ZkClient(ZK_SERVICE, ZK_SESSION_TIME_OUT,
168 ZK_CONNECTION_TIME_OUT, new SerializableSerializer());
169             }
170             //创建 ZK 命名空间/当前部署应用 APP 命名空间
171             boolean exist = zkClient.exists(ROOT_PATH);
172             if (!exist) {
173                 zkClient.createPersistent(ROOT_PATH, true);
174             }
175             //创建服务提供者节点
176             exist = zkClient.exists((ROOT_PATH));
177             if (!exist) {
178                 zkClient.createPersistent(ROOT_PATH);
179             }
180
181             //创建服务消费者节点
182             String serviceNode = invoker.getServiceItf().getName();
183             String servicePath = ROOT_PATH + "/" + serviceNode +
INVOKER_TYPE;

```

```

184         exist = zkClient.exists(servicePath);
185         if (!exist) {
186             zkClient.createPersistent(servicePath);
187         }
188
189         //创建当前服务器节点
190         String localIp = IPHelper.localIp();
191         String currentServiceIpNode = servicePath + "/" + localIp;
192         exist = zkClient.exists(currentServiceIpNode);
193         if (!exist) {
194             //注意，这里创建的是临时节点
195             zkClient.createEphemeral(currentServiceIpNode);
196         }
197     }
198 }
199
200
201
202 //利用 ZK 自动刷新当前存活的服务提供者列表数据
203 private void refreshActivityService(List<String> serviceIpList) {
204     if (serviceIpList == null) {
205         serviceIpList = Lists.newArrayList();
206     }
207
208     Map<String, List<ProviderService>> currentServiceMetaDataMap =
209 Maps.newHashMap();
210     for (Map.Entry<String, List<ProviderService>> entry :
211 providerServiceMap.entrySet()) {
212         String key = entry.getKey();
213         List<ProviderService> providerServices = entry.getValue();
214
215         List<ProviderService> serviceMetaDataModelList =
216 currentServiceMetaDataMap.get(key);

```

```

217         if (serviceMetaDataModelList == null) {
218             serviceMetaDataModelList = Lists.newArrayList();
219         }
220
221         for (ProviderService serviceMetaData : providerServices) {
222             if (serviceIpList.contains(serviceMetaData.getServerIp())) {
223                 serviceMetaDataModelList.add(serviceMetaData);
224             }
225         }
226         currentServiceMetaDataMap.put(key,
serviceMetaDataModelList);
227     }
228     providerServiceMap.putAll(currentServiceMetaDataMap);
229 }
230
231
232 private void refreshServiceMetaDataMap(List<String> serviceIpList)
{
233     if (serviceIpList == null) {
234         serviceIpList = Lists.newArrayList();
235     }
236
237     Map<String, List<ProviderService>> currentServiceMetaDataMap =
238 com.google.common.collect.Maps.newHashMap();
239 for (Map.Entry<String, List<ProviderService>> entry :
240 serviceMetaDataMap4Consume.entrySet()) {
241     String serviceItfKey = entry.getKey();
242     List<ProviderService> serviceList = entry.getValue();
243
244     List<ProviderService> providerServiceList =
245 currentServiceMetaDataMap.get(serviceItfKey);
246     if (providerServiceList == null) {
247         providerServiceList = Lists.newArrayList();

```

```

248         }
249
250         for (ProviderService serviceMetaData : serviceList) {
251             if
(serviceIpList.contains(serviceMetaData.getServerIp())) {
252                 providerServiceList.add(serviceMetaData);
253             }
254         }
255         currentServiceMetaDataMap.put(serviceItfKey,
providerServiceList);
256     }
257     serviceMetaDataMap4Consume.putAll(currentServiceMetaDataMap);
258     System.out.println("serviceMetaDataMap4Consume:" +
259 JSON.toJSONString(serviceMetaDataMap4Consume));
260 }
261
262
263     private          Map<String,          List<ProviderService>>
fetchOrUpdateServiceMetaData() {
264         final Map<String, List<ProviderService>> providerServiceMap =
265 com.google.common.collect.Maps.newConcurrentMap();
266         //连接 ZK
267         synchronized (RegisterCenter.class) {
268             if (zkClient == null) {
269                 zkClient = new ZkClient(ZK_SERVICE, ZK_SESSION_TIME_OUT,
270 ZK_CONNECTION_TIME_OUT, new SerializableSerializer());
271             }
272         }
273
274         //从 ZK 获取服务提供者列表
275         String providePath = ROOT_PATH;
276         List<String>          providerServices          =
zkClient.getChildren(providePath);

```



```

277
278     for (String serviceName : providerServices) {
279         String servicePath = providePath + "/" + serviceName +
PROVIDER_TYPE;
280         List<String> ipPathList = zkClient.getChildren(servicePath);
281         for (String ipPath : ipPathList) {
282             String serverIp = StringUtils.split(ipPath, "|")[0];
283             String serverPort = StringUtils.split(ipPath, "|")[1];
284
285             List<ProviderService> providerServiceList =
286 providerServiceMap.get(serviceName);
287             if (providerServiceList == null) {
288                 providerServiceList = Lists.newArrayList();
289             }
290             ProviderService providerService = new ProviderService();
291
292             try {
293
providerService.setServiceItf(ClassUtils.getClass(serviceName));
294             } catch (ClassNotFoundException e) {
295                 throw new RuntimeException(e);
296             }
297             providerService.setServerIp(serverIp);
298
providerService.setServerPort(Integer.parseInt(serverPort));
299             providerServiceList.add(providerService);
300
301             providerServiceMap.put(serviceName,
providerServiceList);
302         }
303
304         //监听注册服务的变化,同时更新数据到本地缓存
305         zkClient.subscribeChildChanges(servicePath, new

```

```

IZkChildListener() {
    306         @Override
    307         public void handleChildChange(String parentPath,
    308 List<String> currentChilds) throws Exception {
    309             if (currentChilds == null) {
    310                 currentChilds = Lists.newArrayList();
    311             }
    312             currentChilds = Lists.newArrayList(Lists.transform
    313 (currentChilds, new Function<String, String>() {
    314                 @Override
    315                 public String apply(String input) {
    316                     return StringUtils.split(input, "|")[0];
    317                 }
    318             }));
    319             refreshServiceMetaDataMap(currentChilds);
    320         }
    321     });
    322 }
    323     return providerServiceMap;
    324 }
    325 }

```

第 35~42 行，从 `properties` 文件获取 ZooKeeper 配置信息，依次为主机地址列表 `ZK_SERVICE`、会话超时时间 `ZK_SESSION_TIME_OUT`、连接超时时间 `ZK_CONNECTION_TIME_OUT`。

第 58~141 行，方法 `registerProvider()` 完成服务提供者信息注册服务中心功能。首先加同步锁 `synchronized (RegisterCenter.class)`，防止重复注册。

67~77 行将服务提供者列表 `serviceMetaData` 转换为 `providerServiceMap`，其中，Key 为服务提供者接口，value 为服务提供者服务方法列表。79 行~114 行，连接 `ZkClient`，将服务提供者信息作为 ZooKeeper 子节点写入 ZooKeeper，需要注意的是最后的子节点 `ip|port` 作为临时节点写入 ZooKeeper，`ZkClient.createEphemeral (currentServiceIpNode)`，原因是

ZooKeeper 会监听服务端的存活，一旦服务端下线，该临时节点会被自动删除，同时推送给服务消费端，从而达到该服务提供者信息自动下线的目的。

第 117~141 行，将节点路径注册监听器，一旦节点有变化，将自动更新本地服务提供者缓存信息 `providerServiceMap`。

第 151~156 行方法 `initProviderMap()` 完成初始化服务调用方本地缓存 `serviceMetaDataMap4Consume` 的功能。

第 165~236 行方法 `registerInvoker()` 完成服务调用方注册服务中心的功能。实现思路与之前服务提供方类似，在此不再详述。

5.4 本章小结

本章介绍了注册中心的概念及实现原理。详细介绍了 ZooKeeper 的安装、Java API 操作及如何实现注册中心。

第 6 章

分布式服务框架底层通信实现

6.1 Java I/O 模型及 I/O 类库的进化

通信的本质是 I/O，本节回顾一下 I/O 模型及常用的 Java I/O 知识。

6.1.1 Linux 下实现的 I/O 模型

因为程序运行在操作系统上，编程语言实现的 I/O 操作 API 最终依赖于操作系统的 I/O 实现。Linux 操作系统目前占服务器市场大部分份额，下面一起了解一下 Linux 操作系统实现的几种 I/O 模型及其特点。

在这之前，先理清阻塞、非阻塞、同步、异步这 4 个概念。

- ◎ 阻塞：调用方发起调用请求，在没有返回结果之前，调用方线程被挂起，处于一直等待状态。
- ◎ 非阻塞：非阻塞和阻塞的概念相对应，调用方发起调用请求，当前线程不会等待

挂起，而会立刻返回。后续可以通过轮询等手段来获取调用结果状态。

- ◎ 同步：所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。
- ◎ 异步：异步的概念和同步相对。当一个异步过程调用发出后，调用者不会立刻得到结果，通过回调等措施来处理这个调用。

Linux 下实现了 5 种 I/O 模型。

(1) 阻塞 I/O 模型：默认情况下，所有的文件操作都是阻塞的。在进程空间中调用 `recvfrom`，其系统调用直到数据包到达且被复制到应用进程的缓冲区中或者发生错误才返回，在此期间会一直等待，进程在从调用 `recvfrom` 开始到它返回的整段时间内都是被阻塞的，如图 6-1 所示。

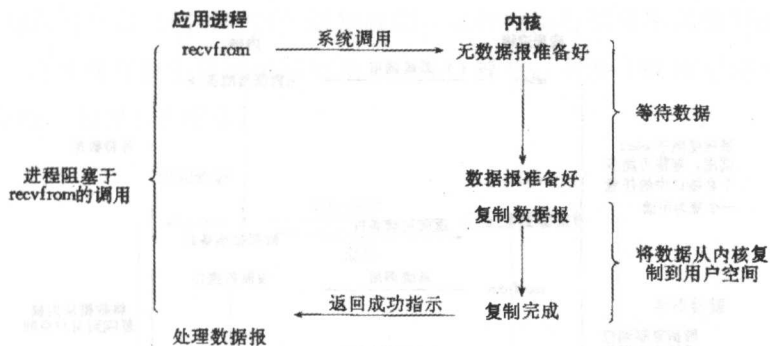


图 6-1 阻塞 I/O 模型

(2) 非阻塞 I/O 模型：进程把一个套接口设置成非阻塞是在通知内核。当所请求的 I/O 操作不能满足要求的时候，不把本进程投入睡眠，而是返回一个错误。也就是说当数据没有到达时并不等待，而是以一个错误返回，如图 6-2 所示。

(3) I/O 复用模型：Linux 提供 `select/poll`，进程通过将一个或多个 `fd` 传递给 `select` 或 `poll` 系统调用，阻塞在 `select`；这样 `select/poll` 可以帮我们侦测许多 `fd` 是否就绪。但是 `select/poll` 是顺序扫描 `fd` 是否就绪的，而且支持的 `fd` 数量有限。Linux 还提供了 `epoll`

系统调用，`epoll` 基于事件驱动方式，而不是顺序扫描，当有 `fd` 就绪时，立即回调函数 `rollback`，如图 6-3 所示。

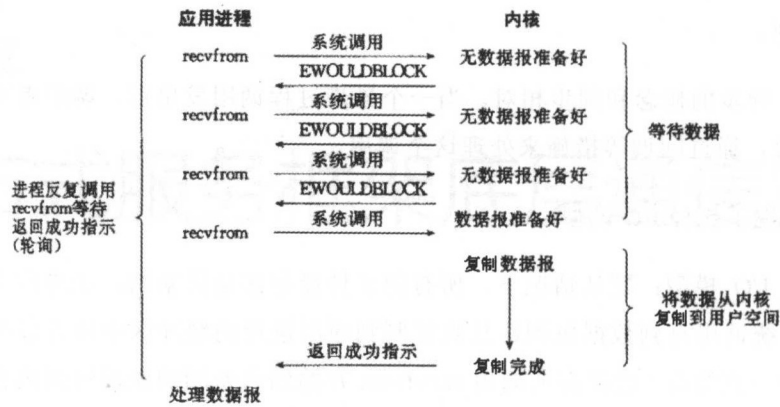


图 6-2 非阻塞 I/O 模型

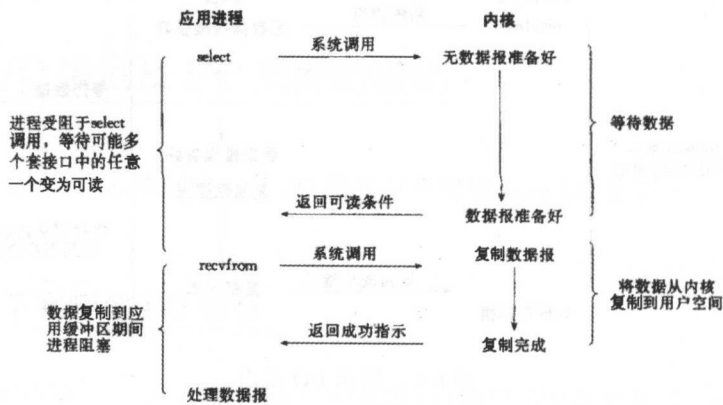


图 6-3 I/O 复用模型

(4) 信号驱动异步 I/O 模型：首先开启套接口信号驱动 I/O 功能，并通过系统调用 `sigaction` 安装一个信号处理函数（此系统调用立即返回，进程继续工作，它是非阻塞的）。当数据报准备好被读时，就为该进程生成一个 `SIGIO` 信号。随即可以在信号处理程序中调用 `recvfrom` 来读数据报，并通知主循环数据已准备好被处理。也可以通知主循环，让它来读数据报，如图 6-4 所示。

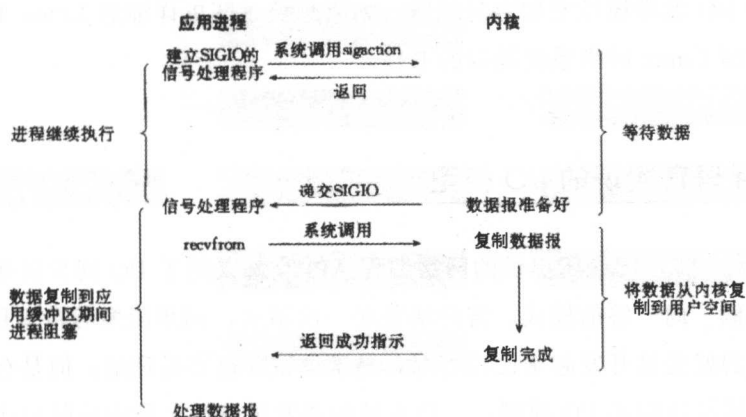


图 6-4 信号驱动异步 I/O 模型

(5) 异步 I/O 模型：告知内核启动某个操作，并让内核在整个操作完成后（包括将数据从内核复制到用户自己的缓冲区）通知我们。这种模型与信号驱动模型的主要区别是信号驱动 I/O 由内核通知我们何时可以启动一个 I/O 操作；异步 I/O 模型由内核通知我们 I/O 操作何时完成，如图 6-5 所示。

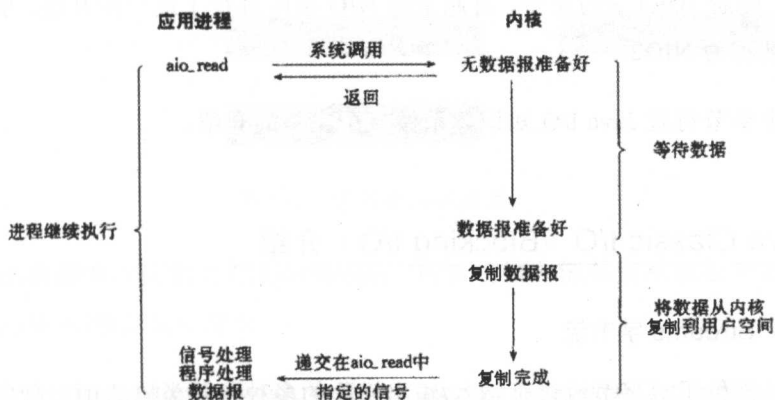


图 6-5 异步 I/O 模型

以上^①对 Linux 所实现的 I/O 模型做了一个简要的介绍，目的是为使下面介绍的 Java

① 以上参考资料 <http://blog.csdn.net/colzer/article/details/8169075>, <http://www.iteye.com/topic/868702>。

语言所实现的 I/O 编程模型更加容易理解，如果想要了解更详细的 Linux I/O 信息，请另行参阅其他讲解 Linux 网络系统编程的书籍。

6.1.2 Java 语言实现的 I/O 模型

在 JDK1.4 之前，Java 所提供的网络编程 API 全部采用了 I/O 同步阻塞模型。具体到编程实现，类似一问一答的模式。客户端发起一次请求，同步阻塞等待调用结果的返回。这种编程模型的好处是开发起来比较容易，易于调试，也容易理解，但是存在严重的性能问题，因为采用同步阻塞 I/O 模型，一旦系统的并发量变高，且响应时间比较大，服务端只能通过升级硬件或者扩容机器来解决吞吐量跟不上的问题。可以说，此时的 Java 在支持大型应用服务器后端服务能力上无法与 C、C++ 竞争。

为了解决这个弊端，JDK1.4 引入了非阻塞 I/O（NIO）类库，自此 Java 语言可以支持多路复用 I/O 模型。由此，也产生了一批著名的 Java NIO 开源网络通信框架，比如 Mina、Netty、Grizzly。

再后来，随着 JDK1.7 的发布，对原来的 NIO 类库进行了进一步升级，引入了异步 I/O 编程类库，被称为 NIO2。

下面几个章节将就 Java I/O 知识体系做一个完整的介绍。

6.1.3 Java Classic I/O（Blocking I/O）介绍

6.1.3.1 Streams 字节流

java.io 包提供了对字节流进行输入/输出操作的多种包装类以适用多种应用场景。下面对输入/输出字节流分别进行介绍。

常用输出和输入字节流简要类如图 6-6 和图 6-7 所示。

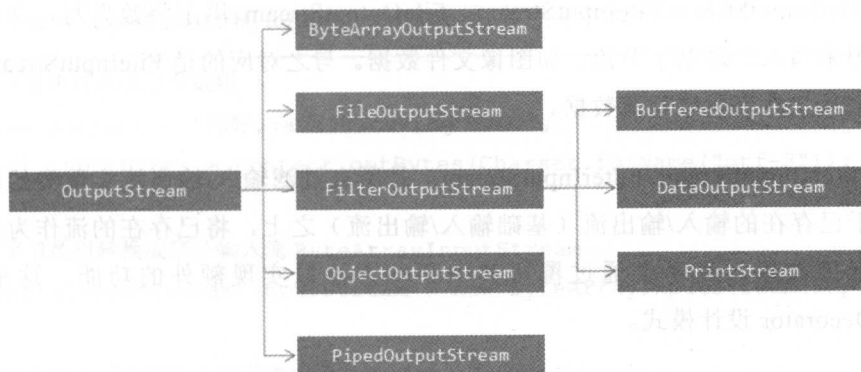


图 6-6 常用输出流类图

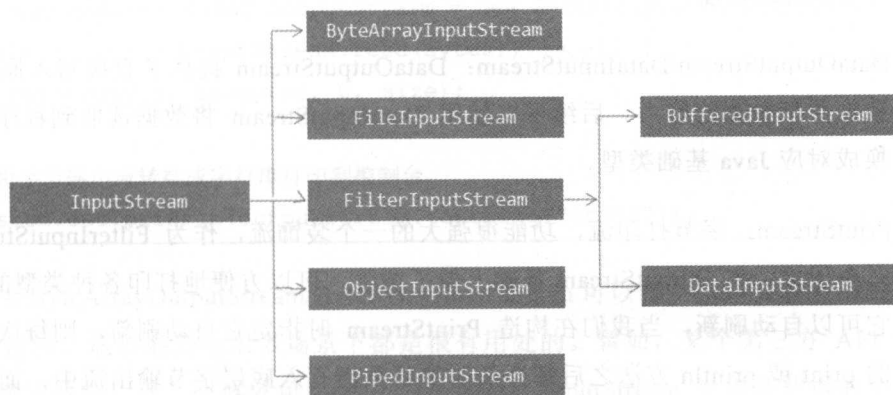


图 6-7 常用输入流类图

所有的输出流都继承抽象类 `OutputStream`，所有的输入流都继承抽象类 `InputStream`。下面介绍常用的输入/输出流实现类。

- ◎ **ByteArrayOutputStream/ByteArrayInputStream**: `ByteArrayOutputStream` 可以将数据写入字节数组，随着数据的写入能够自动扩容，可以通过调用 `toByteArray()` 方法获取写入的字节数组或者通过 `toString()` 方法获取写入的字节的字符串表示。无须调用 `close()` 方法进行关闭。与之对应的是 `ByteArrayInputStream`、`ByteArrayInputStream`，可以将字节数组转换成输入流。

- ◎ **FileOutputStream/FileInputStream:** `FileOutputStream` 用于将数据写入文件。一般用来写入二进制字节流，如图像文件数据。与之对应的是 `FileInputStream` 类，用来从文件读取字节流数据。
- ◎ **FilterOutputStream/FilterInputStream:** 是所有过滤输入/输出流实现类的超类。位于已存在的输入/输出流（基础输入/输出流）之上，将已存在的流作为其基本数据接收器，其子类通过覆写其中某些方法以实现额外的功能。这里使用了 Decorator 设计模式。
- ◎ **BufferedOutputStream/BufferedInputStream:** 先将字节数据写入该缓冲类，再一次性输出。将单字节操作转变为批量操作字节数组，避免了逐个字节处理操作，提高了 I/O 处理性能。
- ◎ **DataOutputStream/DataInputStream:** `DataOutputStream` 提供了直接写入原生 Java 数据类型数据的能力。后续可以使用 `DataInputStream` 将数据读取到程序中并转换成对应 Java 基础类型。
- ◎ **PrintStream:** 字节打印流，功能很强大的一个装饰流，作为 `FilterInputStream` 的一个子类，在 `OutputStream` 基础上做了增强，可以方便地打印各种类型的数据。它可以自动刷新，当我们在构造 `PrintStream` 时指定它自动刷新，则每次调用它的 `print` 或 `println` 方法之后都会及时地将数据写入底层字节输出流中，而不用手动调用 `flush` 去刷新。还可以在构造的时候指定编码，使得字节以我们想要的编码形式写入底层字节输出流中，进而保存到存储介质中。还有一个特性就是 `PrintStream` 的方法从不抛出 `IOException`。
- ◎ **ObjectOutputStream/ObjectInputStream:** Java 对象字节输入/输出流，一般用来实现 Java 的序列化功能。
- ◎ **PipedOutputStream/PipedInputStream:** 通过管道读写字节流。

注意：I/O 流参考 http://blog.csdn.net/crave_shy/article/details/16866755。

下面通过具体的代码示例来说明上述 I/O 字节流的具体使用。

ByteArrayOutputStream/ByteArrayInputStream 使用示例如下。

```
//将字符串转换成字节数组
String content = "你好,java Blocking I/O!";
byte[] inputBytes = content.getBytes(Charset.forName("utf-8"));

//将字节数组转换成字节输入流 ByteArrayInputStream
ByteArrayInputStream inputStream = new ByteArrayInputStream(inputBytes);

//将字节输入流数据写入字节输出流 ByteArrayOutputStream
byte[] bytes = new byte[1024];
int size = 0;
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
while ((size = inputStream.read(bytes)) != -1) {
    outputStream.write(bytes, 0, size);
}
//将字节输出流转换成字符串打印到控制台
System.out.println(outputStream.toString("utf-8"));
```

使用 ByteArrayOutputStream/ByteArrayInputStream 可以将字符串或者字节数组转换成输入/输出流。这种能力在很多场景下都是很有用处的。譬如，某个第三方 API 使用字节流对外输出数据，这个时候就可以使用 ByteArrayOutputStream 对象将获得的字节流暂时保存到内存，不必保存到磁盘。

FileOutputStream/FileInputStream 使用示例如下。

```
FileInputStream inputStream = null;
FileOutputStream outputStream = null;
//定义源文件与目标文件
File srcFile = new File("/Users/liyebing/src.txt");
File targetFile = new File("/Users/liyebing/target.txt");

try {
    //实例化文件输入流与文件输出流
```

```
        inputStream = new FileInputStream(srcFile);
        outputStream = new FileOutputStream(targetFile);

        //通过文件输入流读取源文件内容，并写入目标文件
        int byt;
        while ((byt = inputStream.read()) != -1) {
            outputStream.write(byt);
        }

    } finally {
        if (inputStream != null) {
            inputStream.close();
        }
        if (outputStream != null) {
            outputStream.close();
        }
    }
}
```

从类名可以知晓，`FileOutputStream/FileInputStream` 专门用于文件 I/O 操作。因操作对象为字节流，故 `FileOutputStream/FileInputStream` 更适用于图片等二进制文件操作。

使用 `BufferedOutputStream/BufferedInputStream` 提高 I/O 性能示例如下。

```
FileInputStream inputStream = null;
FileOutputStream outputStream = null;
BufferedInputStream bufferedInput = null;
BufferedOutputStream bufferedOutput = null;

//定义源文件与目标文件
File srcFile = new File("/Users/liyebing/src.txt");
File targetFile = new File("/Users/liyebing/target.txt");

try {
    //实例化文件输入流与文件输出流
```

```

        inputStream = new FileInputStream(srcFile);
        bufferedInput = new BufferedInputStream(inputStream);

        outputStream = new FileOutputStream(targetFile);
        bufferedOutput = new BufferedOutputStream(outputStream);

        //通过缓冲输入流读取源文件内容，并写入缓冲输出流，最终写入文件
        byte[] buff = new byte[1024];
        int byt;
        while ((byt = bufferedInput.read(buff, 0, buff.length)) != -
1){
            bufferedOutput.write(buff, 0, byt);
        }
        bufferedOutput.flush();
    } finally {
        if (inputStream != null) {
            inputStream.close();
        }
        if (outputStream != null) {
            outputStream.close();
        }
    }
}

```

BufferedOutputStream/BufferedInputStream 使用了装饰器模式（Decorator）。在 **FileOutputStream/FileInputStream** 基础上增加了字节流缓存能力，使得对字节数组的读取写入能够批量进行，对比 **FileOutputStream/FileInputStream** 大幅度提高了 I/O 操作效率与性能。

DataOutputStream/DataInputStream 使用示例如下。

```

String fileName = "/Users/liyebing/data.txt";

//将 Java 原生类型数据通过 DataOutputStream 写入文件
FileOutputStream fout = new FileOutputStream(fileName);

```

```
DataOutputStream dos = new DataOutputStream(fout);

dos.writeInt(2017);
dos.writeUTF("你好,java Blocking I/O!");
dos.writeBoolean(true);

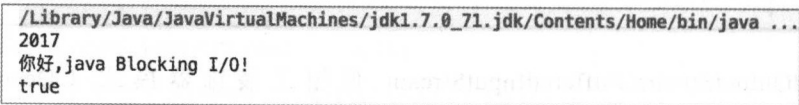
dos.close();
fout.close();

//使用 DataInputStream 从文件中按照写入顺序读取 Java 原生类型数据
FileInputStream fin = new FileInputStream(fileName);
DataInputStream dis = new DataInputStream(fin);

System.out.println(dis.readInt());
System.out.println(dis.readUTF());
System.out.println(dis.readBoolean());

dis.close();
fin.close();
```

运行结果如图 6-8 所示，可以看到按照写入的顺序打印出 Java 的原生类型数据。



```
/Library/Java/JavaVirtualMachines/jdk1.7.0_71.jdk/Contents/Home/bin/java ...
2017
你好,java Blocking I/O!
true
```

图 6-8 Java 原生型数据

`DataOutputStream/DataInputStream` 能够对 Java 原生类型直接写入再按写入顺序直接读取。常用于网络数据传输过程中的写入与读取。

`PrintStream` 使用示例如下。

```
File file = new File("/Users/liyebing/p.txt");
PrintStream printStream = new PrintStream(file);
```

```
//将内容写入文件
printStream.println("你好,java Blocking I/O!");
printStream.close();
```

PrintStream 常用于日志输出组件的实现，无须抛出 I/O 异常的场景。

ObjectOutputStream/ObjectInputStream 使用示例如下。

```
User user = new User();
user.setEmail("kongxuan@163.com");
user.setName("kongxuan");

//将 User 对象序列化到文件
FileOutputStream fout = new FileOutputStream("user.txt");
ObjectOutputStream oout = new ObjectOutputStream(fout);
oout.writeObject(user);
oout.close();
fout.close();

//从 user.txt 文件中反序列化得到 User 对象
FileInputStream fin = new FileInputStream("user.txt");
ObjectInputStream oin = new ObjectInputStream(fin);
User newUser = (User) oin.readObject();
System.out.println("email:" + newUser.getEmail() + " name:" +
newUser.getName());
oin.close();
fin.close();
```

ObjectOutputStream/ObjectInputStream 常用于 Java 对象的序列化/反序列化或者网络数据的写入与读取。

6.1.3.2 Writer 与 Reader 字符流

Java Stream 相关的类用来处理字节流，但是不适合用来处理字符流。因为一个字节 8 bit，而一个字符是 16 bit，字符串由字符组成，字符串类型天然处理的是字符而不是字

节。更重要的是，字节流无法知道字符集及其字符编码。java.io 包提供了用来处理字符流的抽象类 `Writer` 与 `Reader` 及相应的子类，常用的字符输出和输入字符流如图 6-9 和图 6-10 所示。

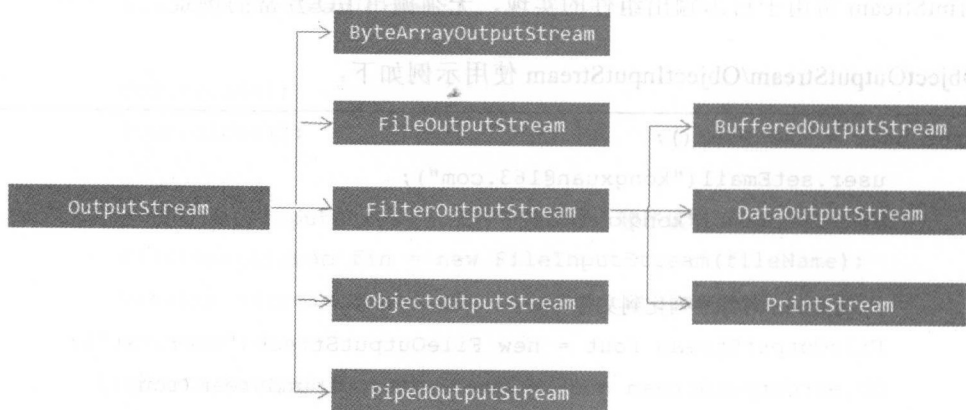


图 6-9 常用字节输出流

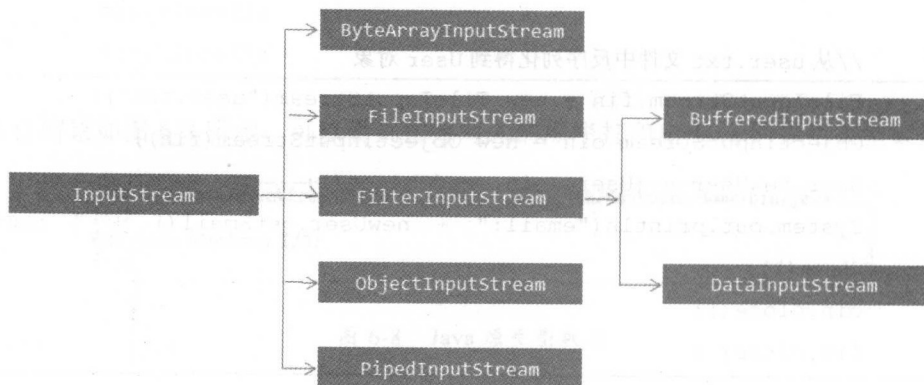


图 6-10 常用字节输入流

所有的输出流都继承抽象类 `Writer`，所有的输入流都继承抽象类 `Reader`。下面介绍常用的输入输出流实现类。

- ◎ `BufferedWriter/BufferedReader`：先将字符数据写入或者读取到缓冲区，再一次性处

理,相对于逐个字符处理,提高了 I/O 处理性能。可类比字节流 `BufferedOutputStream/BufferedReader`。

- ◎ `CharArrayWriter/CharArrayReader`: `CharArrayWriter` 提供了一个字符类型数组缓存,当写入数据的时候,缓冲区自动增长。可以使用 `toCharArray()` 方法获取字符数组或者使用 `toString()` 方法获取字符串。类似地, `CharArrayReader` 提供了字符输入流的缓存数组。这两个类可类比字节流 `ByteArrayOutputStream/ByteArrayInputStream`。
- ◎ `OutputStreamWriter/InputStreamReader`: `InputStreamReader` 是字符流 `Reader` 的子类,是字节输入流通向字符流的桥梁。你可以在构造器中重新指定字符集,如果不指定将采用底层操作系统的默认字符集。类似, `OutputStreamWriter` 是字符流 `Writer` 的子类,是字符输出流通向字节流的桥梁。在调用 `write()` 方法时会调用编码转换器进行编码。
- ◎ `FileWriter/FileReader`: 分别是 `OutputStreamWriter/InputStreamReader` 的子类,提供了以字符为单位读写文件的能力。在处理文本类型文件时,相对于 `FileOutputStream/FileInputStream` 具有更好的读写性能。
- ◎ `StringWriter/StringReader`: 将字符串 `String` 类型的数据适配到了 `Writer` 与 `Reader` 操作。
- ◎ `PipedWriter/PipedReader`: 通过管道读写字符流,类比 `PipedOutputStream/PipedInputStream`。
- ◎ `PrintWriter`: 除了提供 `PrintStream` 中的所有 `print` 方法,还提供了格式化输出字符串的能力。其方法不会抛出 I/O 异常。

下面通过具体的代码示例来说明上述 I/O 字符流的具体使用。

`BufferedWriter/BufferedReader` 及 `FileWriter/FileReader` 使用示例如下。

```
FileReader fr = new FileReader("src.txt");  
FileWriter fw = new FileWriter("target.txt");
```

```
//将字符输入/输出流用 BufferedReader/BufferedWriter 包装起来
BufferedReader bufReader = new BufferedReader(fr);
BufferedWriter bufWriter = new BufferedWriter(fw);

//利用 BufferedReader/BufferedWriter 实现逐行读写，提高 I/O 性能
String line = null;
while ((line = bufReader.readLine()) != null) {
    bufWriter.write(line);
    bufWriter.newLine();
}
bufWriter.flush();
bufReader.close();
bufWriter.close();
```

FileWriter/FileReader 一般用来操作文本文件，**BufferedWriter/BufferedReader** 使用了装饰器模式（Decorator），增加了对字符流操作缓存能力，使其能够批量读写字符流，提高了 I/O 操作的效率与性能。

CharArrayWriter/CharArrayReader 使用示例如下。

```
//将字符串转换成字符数组
String content = "你好!java Blocking I/O!";
//将字符数组转换成字符输入流 CharArrayReader
CharArrayReader charReader = new CharArrayReader(content.toCharArray());
//将字符输入流数据写入字符输出流 CharArrayWriter
char[] chars = new char[1024];
int size = 0;
CharArrayWriter charWriter = new CharArrayWriter();
while ((size = charReader.read(chars)) != -1) {
    charWriter.write(chars, 0, size);
}

//获取字符串并打印到控制台
```

```

System.out.println(charWriter.toString());
//获取字符数组并打印到控制台
char[] charArray = charWriter.toCharArray();
for (char c : charArray) {
    System.out.println(c);
}

```

利用 `CharArrayWriter/CharArrayReader` 能够将字符串或者字符数组转换为字符流。这种能力在很多场景下都是很有用处的。譬如，某个第三方 API 使用字符流对外输出数据，这个时候就可以使用 `CharArrayWriter` 对象将获得的字节流暂时保存到内存，不必保存到磁盘。

`OutputStreamWriter/InputStreamReader` 使用示例如下。

```

//创建文件字节输入流
FileInputStream inputStream = new FileInputStream("src.txt");
//利用桥梁 InputStreamReader 将文件字节输入流 inputStream 转换成字符输入流
InputStreamReader inputStreamReader = new InputStreamReader(inputStream, "utf-8");
//利用 BufferedReader 包装字符输入流 inputStreamReader，提高性能
BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

//创建文件字节输出流
FileOutputStream outputStream = new FileOutputStream("target.txt");
//利用桥梁 outputStreamWriter 将文件字节输出流 outputStream 转换成字符输出流
OutputStreamWriter outputStreamWriter = new OutputStreamWriter(outputStream);
//利用 BufferedWriter 包装字符输出流 outputStreamWriter，提高性能
BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);

//将文件 src.txt 文本内容写入 target.txt
String line = null;

```

```
while ((line = bufferedReader.readLine()) != null) {  
    bufferedWriter.write(line);  
    bufferedWriter.newLine();  
}  
bufferedWriter.flush();  
  
bufferedReader.close();  
bufferedWriter.close();  
inputStream.close();  
outputStream.close();
```

OutputStreamWriter/InputStreamReader 作为字节流与字符流之间的桥梁，能够将字节流转换为字符流。比如，第三方接口返回的数据是字节流形式的文本，为了提高操作性能及操作便捷性，可以使用 **OutputStreamWriter/InputStreamReader** 将字节流转换为字符流。

PrintWriter 使用示例如下。

```
File file = new File("target.txt");  
PrintWriter pw = new PrintWriter(file);  
//将内容格式化输出到文件 target.txt  
pw.format("你好,%s %s %s %s", "java", "Blocking", "I/O", "!");  
pw.flush();  
pw.close();
```

PrintWriter 用于需要格式化输出的场景。

6.1.3.3 Java blocking I/O 网络通信实现

在 6.1.1 节我们介绍了 Linux 下的 I/O 模型，其中有对阻塞模型的介绍。本节我们将介绍 Java 语言中 I/O 阻塞模型的实现。

在网络上，数据按照有限大小的数据报（datagram）进行传输，每个数据报分为两部分：**header** 首部和 **payload** 有效载荷。首部包含数据报目的地址与端口、来源地址与端口、检测数据是否被破坏的校验和用于保证可靠传输的各种其他管理信息。因为数据报大小有

限，需要将数据分解为多个包，再在目的地重新组合，在传输过程中，可能有数据丢失或者损坏，这个时候需要数据包重传。还有一种可能是数据包乱序到达，需要进行重排序。幸运的是，你不需要自己来完成这些繁重冗杂的工作，Socket 对开发人员封装了这些网络底层细节。

Java Socket 实现分为客户端 Socket 与服务端 ServerSocket。

客户端 Socket 的使用方式如下。

- ◎ 创建 Socket 对象，使用创建的 Socket 连接远程主机。
- ◎ 建立连接后，从 Socket 得到输入流与输出流，Socket 是全双工通道，可以使用这两个流与服务器之间相互发送数据。

服务端 ServerSocket 的使用方式如下。

- ◎ 绑定一个特定的端口创建 ServerSocket 对象。
- ◎ 使用 ServerSocket 的 accept()方法监听这个端口的请求连接，accept()会一直阻塞直到通过某个请求连接与客户端建立连接，此时 accept()将返回客户端与服务端的连接的 Socket 对象。
- ◎ 通过 Socket 对象的 getInputStream()与 getOutputStream()方法获得与客户端通信的输入流与输出流，进行通信交互。
- ◎ 完成交互后关闭连接。

下面通过一个代码示例来具体演示上述流程。

BIOEchoClient.java 客户端 Socket 发起请求代码如下：

```
import java.io.*;
import java.net.Socket;

public class BIOEchoClient {
```

```

public static void main(String[] args) throws IOException {
    int port = 8082;
    String serverIp = "127.0.0.1";
    Socket socket = null;
    BufferedReader reader = null;
    BufferedWriter writer = null;
    try {
        //创建 Socket 对象，并连接远程主机
        socket = new Socket(serverIp, port);
        //建立连接后，从 Socket 得到输入流与输出流，可以使用这两个流与服务器之间相互
        发送数据
        reader = new BufferedReader(new InputStreamReader
(socket.getInputStream()));
        writer = new BufferedWriter(new OutputStreamWriter
(socket.getOutputStream()));
        //向服务端发送数据
        writer.write("Hello,Block IO.\n");
        writer.flush();
        //获取服务端返回的数据
        String echo = reader.readLine();
        System.out.println("echo:" + echo);
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e1) {
                e1.printStackTrace();
            }
        }
        if (socket != null) {
            try {
                socket.close();
            } catch (IOException e3) {

```

```

        e3.printStackTrace();
    }
}
}
}
}
}

```

BIOEchoService 服务端处理客户端请求代码如下:

```

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class BIOEchoService {

    //服务端处理业务逻辑线程池
    private static final ExecutorService
    executor= Executors.newCachedThreadPool();

    public static void main(String[] args) throws IOException {
        int port = 8082;
        ServerSocket serverSocket = null;
        try {
            //绑定一个特定的端口创建 ServerSocket 对象
            serverSocket = new ServerSocket(port);
            Socket socket = null;
            while (true) {
                //使用 ServerSocket 的 accept() 方法监听这个端口的请求连接
                socket = serverSocket.accept();
                executor.submit(new BIOEchoServerHandler(socket));
            }
        } finally {

```

```
        if (serverSocket != null) {  
            serverSocket.close();  
        }  
    }  
}  
}
```

BIOEchoServerHandler 服务端处理业务逻辑的代码：

```
public class BIOEchoServerHandler implements Runnable {  
    private Socket socket;  
    public BIOEchoServerHandler(Socket socket) {  
        this.socket = socket;  
    }  
  
    public void run() {  
        BufferedReader reader = null;  
        BufferedWriter writer = null;  
        try {  
            //通过 Socket 对象的 getInputStream() 与 getOutputStream() 方法获得与客  
            户端通信的输入流与输出流  
            reader = new BufferedReader(new InputStreamReader(this.socket.  
getInputStream()));  
            writer = new BufferedWriter(new OutputStreamWriter(this.socket.  
getOutputStream()));  
            while (true) {  
                //获取客户端的数据  
                String line = reader.readLine();  
                if (line == null) {  
                    break;  
                }  
                //将客户端获取到的数据  
                writer.write(line + "\n");  
                writer.flush();  
            }  
        }  
    }  
}
```



```

    }
} catch (Exception e) {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
    if (this.socket != null) {
        try {
            this.socket.close();
        } catch (IOException e3) {
            e3.printStackTrace();
        }
    }
}
}
}
}
}

```

6.1.4 Java Non-blocking I/O (NIO) 介绍

6.1.4.1 Buffer 缓冲区

在 Classic I/O 库中，数据直接面向 Stream 写入或者读取，而在 NIO 库中，数据读取与写入面向的是 Buffer 对象，这种差异使性能得到了巨大提高。打个比方，运输一堆沙子，面向 Stream Classic I/O 的数据读取或者写入好比是使用小铲子一铲子一铲子把沙子铲走，而在 NIO 库面向 Buffer 的数据读取或者写入好比是使用卡车运输沙子，先将沙子装在卡车上，再一次性运走。这里 Buffer 就充当了卡车的角色。

缓冲区实质上是一个数组，java.nio 库中提供了 Buffer 抽象类，基于该抽象类，实现了一系列 Java 基本类型的 Buffer 子类，Buffer 类图如图 6-11 所示。

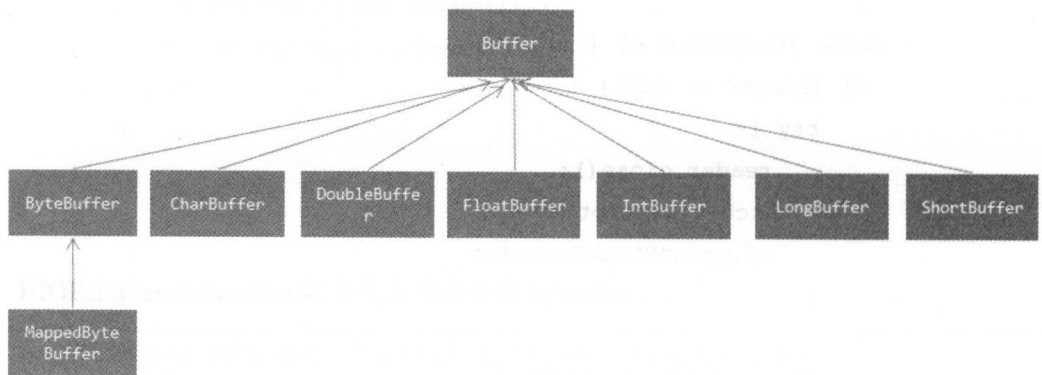


图 6-11 Buffer 类图

所有的缓冲区都具有以下四个属性来提供关于其所包含的数据元素的信息。

- ◎ 容量 (Capacity): 缓冲区能够容纳的数据元素的最大数量。这一容量在缓冲区创建时被设定, 并且永远不能被修改。
- ◎ 上界 (Limit): 缓冲区的第一个不能被读或写的元素。或者说, 缓冲区中现存元素的计数。
- ◎ 位置 (Position): 下一个要被读或写的元素的索引。位置会自动由相应的 `get()` 和 `put()` 函数更新。
- ◎ 标记 (Mark): 一个备忘位置。调用 `mark()` 来设定 `mark=position`。调用 `reset()` 设定 `position=mark`。标记在设定前是未定义的 (undefined)。

这四个属性之间总是遵循以下关系:

$$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$$

下面通过代码示例来演示以上属性的关系。

Buffer 的属性值变化示例如下。

```
Buffer buffer = ByteBuffer.allocate(10);
System.out.println("Capacity:" + buffer.capacity());
```

```

System.out.println("limit:" + buffer.limit());
System.out.println("Position:" + buffer.position());
System.out.println("Remaining:" + buffer.remaining());
System.out.println("设置buffer的limit属性为6");
buffer.limit(6);
System.out.println("limit:" + buffer.limit());
System.out.println("Position:" + buffer.position());
System.out.println("Remaining:" + buffer.remaining());
buffer.position(2);
System.out.println("Position:" + buffer.position());
System.out.println("Remaining:" + buffer.remaining());
System.out.println(buffer);

```

运行结果:

```

Capacity:10
limit:10
Position:0
Remaining:10

```

设置 Buffer 的 limit 属性为 6:

```

limit:6
Position:0
Remaining:6
Position:2
Remaining:4
java.nio.HeapByteBuffer[pos=2 lim=6 cap=10]

```

其中 remaining()方法用来获取 Buffer 中可以读取的数据个数。

Buffer 主要的 API 如下。

- ◎ public final int capacity(): 返回 capacity 的值。
- ◎ public final int position(): 返回 position 的值。

- ◎ `public final Buffer position(int newPosition)`: 设置新的 `position`。
- ◎ `public final int limit()`: 返回 `limit` 的值。
- ◎ `public final Buffer limit(int newLimit)`: 设置新的 `limit`。
- ◎ `public final Buffer mark()`: 将当前的 `position` 值设置为 `mark`。
- ◎ `public final Buffer reset()`: 将 `position` 的值设置为前一次设置的 `mark` 值。
- ◎ `public final Buffer clear()`: 清空 `Buffer`, 将 `position` 值设置为 0, `limit` 设置为 `capacity`, `mark` 值设置为无效。
- ◎ `public final Buffer flip()`: 翻转 `Buffer`, 为读 `Buffer` 做准备。`limit` 设置为当前的 `position`, `position` 值为 0, `mark` 值为无效。
- ◎ `public final Buffer rewind()`: 倒回 `Buffer`, 为重新读取 `Buffer` 做准备。`Position` 值为 0, `mark` 值为无效。
- ◎ `public final int remaining()`: 返回 `Buffer` 中数据个数, 即当前 `position` 与 `limit` 之间的数据。
- ◎ `public final boolean hasRemaining()`: 判断 `Buffer` 是否有数据。

下面通过一个代码示例演示 `Buffer` 常用的 API 的使用。

```
String content = "你好!java Non-Blocking I/O.";
CharBuffer buffer = CharBuffer.allocate(50);
//将字符串内容写入 Buffer
for (int i = 0; i < content.length(); i++) {
    buffer.put(content.charAt(i));
}
//反转 Buffer, 准备读取 Buffer 内容
buffer.flip();

//读取 Buffer 中的数据
```

```
while (buffer.hasRemaining()) {  
    System.out.print(buffer.get());  
}  
  
//倒回读取之前, 准备再次读取  
buffer.rewind();  
System.out.println();  
  
//读取 Buffer 中的数据  
while (buffer.hasRemaining()) {  
    System.out.print(buffer.get());  
}  
System.out.println();  
  
//清空 Buffer, 复位 position, Buffer 可以再次复用  
buffer.clear();  
buffer.put('你').put('好').put('!');  
buffer.flip();  
//再次读取 Buffer 中的数据  
while (buffer.hasRemaining()) {  
    System.out.print(buffer.get());  
}  
}
```

运行结果:

```
你好!java Non-Blocking I/O.  
你好!java Non-Blocking I/O.  
你好!
```

6.1.4.2 Channel 通道

Classic I/O 中的 Stream 是单向的, 通过 OutputStream 实现输出流, InputStream 实现输入流。而 NIO 中的 Channel 是一个全双工通道, 可以通过 Channel 实现同时读取与写入。

上节举例说 **Buffer** 是运输沙子的卡车，那么 **Channel** 就是卡车行驶的道路。而 **Channel** 传输的对象只能是 **Buffer** 对象。与缓冲区不同，通道 API 主要由接口指定。不同的操作系统上通道实现会有很大的差异。

通道的具体实现分为文件读写通道与网络读写通道。最重要的通道实现如下。

- ◎ **FileChannel**：从文件中读写数据。
- ◎ **DatagramChannel**：通过 UDP 协议读写网络数据。
- ◎ **SocketChannel**：通过 TCP/IP 协议读写网络数据，客户端连接通道。
- ◎ **ServerSocketChannel**：**SocketChannel** 对应的服务端通道实现，通过监听新的 TCP/IP 连接，对每一个新的连接创建新的 **SocketChannel**。

下面重点介绍 **FileChannel** 及 **SocketChannel**、**ServerSocketChannel** 的使用。

第一种实现方式：使用 **transferTo** 方法实现文件的复制：

```
FileInputStream fin = new FileInputStream("src.txt");
//从输入流中获取源文件 src.txt 的通道
FileChannel finChannel = fin.getChannel();

FileOutputStream fout = new FileOutputStream("target.txt");
//从输出流中获取目标文件 target.txt 的通道
FileChannel foutChannel = fout.getChannel();

//使用 transferTo API 将文件 src.txt 内容写入 target.txt
finChannel.transferTo(0, finChannel.size(), foutChannel);

//关闭文件流及通道
fin.close();
finChannel.close();
fout.close();
foutChannel.close();
```

第二种实现方式：使用 Channel 的 read 与 write 方法实现文件的复制：

```
FileInputStream fin = new FileInputStream("src.txt");
//从输入流中获取源文件 src.txt 的通道
FileChannel finChannel = fin.getChannel();

FileOutputStream fout = new FileOutputStream("target.txt");
//从输出流中获取目标文件 target.txt 的通道
FileChannel foutChannel = fout.getChannel();

//文件读取内容 Buffer
ByteBuffer buf = ByteBuffer.allocate(1024);
int bytesRead = finChannel.read(buf);
//一次性可能读不完，所以需要循环读取
while (bytesRead != -1) {
    //翻转 Buffer，为下面的读取做准备
    buf.flip();
    while (buf.hasRemaining()) {
        //将读取到的内容写入 target.txt
        foutChannel.write(buf);
    }
    //复位 Buffer，以便再次复用 Buffer
    buf.clear();
    bytesRead = finChannel.read(buf);
}

//关闭文件流及通道
fin.close();
finChannel.close();
fout.close();
foutChannel.close();
```

在 6.1.3.3 节介绍了如何使用 Classic I/O 来实现阻塞模式的网络通信应用程序。除此

之外，我们还可以通过 `SocketChannel`、`ServerSocketChannel` 来实现阻塞模式的网络通信，下面给出相应的代码示例。

EchoServer 服务端代码：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class EchoServer {
    //执行服务端业务逻辑线程池
    private ExecutorService executor = Executors.newCachedThreadPool();

    public static void main(String[] args) throws IOException {
        try {
            //新建服务端 serverSocketChannel
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            //如果 serverSocketChannel 创建成功
            if (serverSocketChannel.isOpen()) {
                //设置为阻塞模式
                serverSocketChannel.configureBlocking(true);
                //设置网络传输参数
                serverSocketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4
* 1024);
                serverSocketChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);
                //绑定服务端 Channel 端口与本地 IP
                serverSocketChannel.bind(new InetSocketAddress("127.0.0.1", 8085));
                while (true) {
                    try {
```



```

        //等待连接客户端的请求
        SocketChannel socketChannel =
serverSocketChannel.accept();
        //提交给线程池处理业务逻辑
        executor.submit(new EchoHandler(socketChannel));
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
} else {
    throw new RuntimeException("server socket channel cannot be
opened!");
}
} catch (IOException ex) {
    throw new RuntimeException(ex);
}
}
}
}

```

EchoHandler 服务端业务处理类:

```

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class EchoHandler implements Runnable {

    private SocketChannel socketChannel;

    private ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

    public EchoHandler(SocketChannel socketChannel){
        this.socketChannel=socketChannel;
    }
}

```

```
public void run() {
    try {
        //读取客户端传输的数据，并原样写入返回给客户端
        while (socketChannel.read(buffer) != -1) {
            buffer.flip();
            socketChannel.write(buffer);
            if (buffer.hasRemaining()) {
                buffer.compact();
            } else {
                buffer.clear();
            }
        }
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

EchoClient 客户端:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;

public class EchoClient {

    public static void main(String[] args) {
        ByteBuffer helloBuffer = ByteBuffer.wrap(" 你好 ,java Blocking
```

```

I/O !".getBytes());

CharBuffer charBuffer;

Charset charset = Charset.defaultCharset();

CharsetDecoder decoder = charset.newDecoder();

try {
    //创建客户端 socketChannel
    SocketChannel socketChannel = SocketChannel.open();
    //如果客户端 socketChannel 创建成功
    if (socketChannel.isOpen()) {
        //设置为阻塞模式
        socketChannel.configureBlocking(true);
        //设置网络传输参数
        socketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 128 *
1024);
        socketChannel.setOption(StandardSocketOptions.SO_SNDBUF, 128 *
1024);
        socketChannel.setOption(StandardSocketOptions.SO_KEEPALIVE,
true);
        socketChannel.setOption(StandardSocketOptions.SO_LINGER, 5);
        //连接服务端
        socketChannel.connect(new InetSocketAddress("127.0.0.1", 8085));
        //如果成功连接服务端
        if (socketChannel.isConnected()) {
            //向服务端发送数据
            socketChannel.write(helloBuffer);
            //创建接受服务端返回数据 ByteBuffer
            ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
            while (socketChannel.read(buffer) != -1) {
                buffer.flip();
                charBuffer = decoder.decode(buffer);
                System.out.println(charBuffer.toString());
                if (buffer.hasRemaining()) {

```

```

        buffer.compact();
    } else {
        buffer.clear();
    }
}
} else {
    throw new RuntimeException("connection cannot be
established!");
}
//关闭连接
socketChannel.close();
} else {
    throw new RuntimeException("socket channel cannot be
opened!");
}
} catch (IOException ex) {
    throw new RuntimeException(ex);
}
}
}
}

```

6.1.4.3 Selector 选择器

Channel 在 Selector 上注册，Selector 通过不断轮询注册在其上的 Channel，能够感知到 Channel 可读或者可写事件。通过这种机制，可以使用一个或者少数几个线程管理大量的网络连接。用较少的线程处理大量的网络连接有很大的好处，可以减少线程之间的切换开销，而且线程本身也需要占用系统资源。在 Linux 系统下，目前 JDK 使用 `epoll()` 系统调用代替了传统的 `select` 实现，使其没有最大连接句柄限制，这意味着可以同时支持上万的客户端网络连接。相对于 Classic I/O，在开发高并发网络后端系统方面，这无疑是一个巨大的进步，如图 6-12 所示。

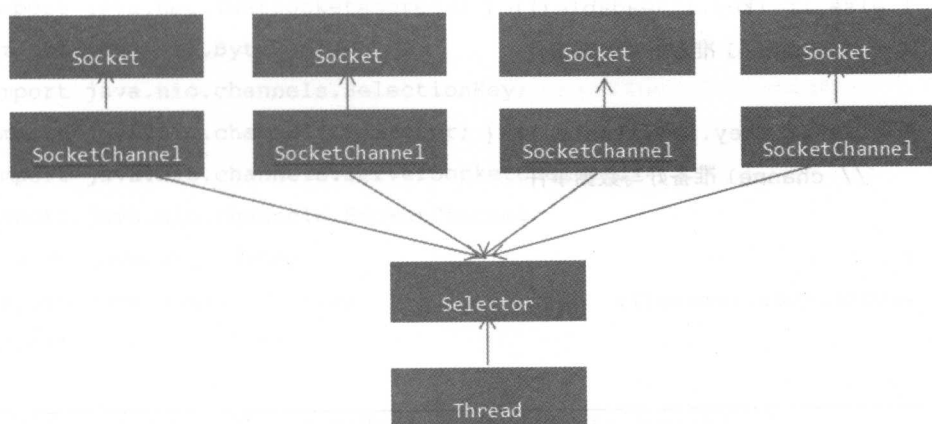


图 6-12 NIO I/O 复用网络通信编程模型

这里有一个完整的示例，打开一个 `Selector`，注册一个通道注册到这个 `Selector` 上（通道的初始化过程略去），然后持续监控这个 `Selector` 的四种事件（接受、连接、读、写）是否就绪，代码如下所示。

```

Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
while(true) {
    int readyChannels = selector.select();
    if(readyChannels == 0) continue;
    Set selectedKeys = selector.selectedKeys();
    Iterator keyIterator = selectedKeys.iterator();
    while(keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();
        if(key.isAcceptable()) {
            // connection accepted 事件

        } else if (key.isConnectable()) {
            // connection established 事件

```

```
        } else if (key.isReadable()) {  
            // channel 准备好读数据事件  
        } else if (key.isWritable()) {  
            // channel 准备好写数据事件  
        }  
        keyIterator.remove();  
    }  
}
```

6.1.4.4 Java I/O 复用网络通信实现

NIO 实现 I/O 复用模型整个编程模型较为复杂。在实际项目中，一般不会直接使用原生 NIO 的 API 构建网络通信程序，而是使用 Mina、Netty、Grizzly 其中之一来构建我们的网络通信服务。所以这里仅了解相关的概念，能够读懂相关的代码即可，不会对 NIO 网络通信部分做深入介绍。

下面给出一个回显服务完整的 NIO 实现的代码。

NIOEchoServer 服务端启动服务代码：

```
public class NIOEchoServer {  
  
    public static void main(String[] args) {  
        int port = 8080;  
  
        NIOEchoServerHandler server = new NIOEchoServerHandler(port);  
        new Thread(server).start();  
    }  
}
```

NIOEchoServerHandler 服务端处理逻辑代码：

```
import java.io.IOException;
```

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Date;
import java.util.Iterator;
import java.util.Set;

public class NIOEchoServerHandler implements Runnable {

    private Selector selector;

    private ServerSocketChannel serverSocketChannel;

    private volatile boolean stop;

    /**
     * 初始化多路复用器，绑定监听端口
     *
     * @param port
     */
    public NIOEchoServerHandler(int port) {
        try {
            //连接队列大小
            int backlog = 1024;
            selector = Selector.open();
            serverSocketChannel = ServerSocketChannel.open();
            serverSocketChannel.configureBlocking(false);
            serverSocketChannel.socket().bind(new InetSocketAddress(port),
backLog);
```

```
        serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
}

public void stop() {
    this.stop = true;
}

public void run() {
    while (!stop) {
        try {
            selector.select(1000);
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> it = selectionKeys.iterator();
            SelectionKey key = null;
            while (it.hasNext()) {
                key = it.next();
                it.remove();

                try {
                    handleInput(key);
                } catch (Exception e) {
                    if (key != null) {
                        key.cancel();
                        if (key.channel() != null) {
                            key.channel().close();
                        }
                    }
                }
            }
        }
    }
}
```



```

    }
    } catch (Throwable e) {
        e.printStackTrace();
    }
}

```

//多路复用器关闭后,所有注册在上面的Channel和Pipe等资源都会被自动注册并关闭,不需要重复释放资源

```

    if (selector != null) {
        try {
            selector.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

private void handleInput(SelectionKey key) throws IOException {
    if (key.isValid()) {
        //处理新接入的请求消息
        if (key.isAcceptable()) {
            ServerSocketChannel ssc = (ServerSocketChannel)
key.channel();

            SocketChannel sc = ssc.accept();
            sc.configureBlocking(false);
            sc.register(selector, SelectionKey.OP_READ);
        }

        //处理数据的读取
        if (key.isReadable()) {
            SocketChannel sc = (SocketChannel) key.channel();
            ByteBuffer readBuffer = ByteBuffer.allocate(1024);
            int readBytes = sc.read(readBuffer);

```

```
//返回值大于 0，读到了字节，对字节进行编解码
if (readBytes > 0) {
    readBuffer.flip();
    byte[] bytes = new byte[readBuffer.remaining()];
    readBuffer.get(bytes);

    String body = new String(bytes, "UTF-8");
    System.out.println("echo content:" + body);

    doWrite(sc, body);
    //返回值为-1，链路已经关闭，需要关闭 SocketChannel，释放资源
} else if (readBytes < 0) {
    key.cancel();
    sc.close();
    //返回值等于 0，没有读取到字节，属于正常场景，忽略
} else {
    ;// 读到 0 字节，忽略
}
}
}

private void doWrite(SocketChannel channel, String response) throws
IOException {
    if (response != null && response.trim().length() > 0) {
        byte[] bytes = response.getBytes();
        ByteBuffer writeBuffer = ByteBuffer.allocate(bytes.length);
        writeBuffer.put(bytes);
        writeBuffer.flip();
        channel.write(writeBuffer);
    }
}
}
```

NIOEchoClient 客户端发起连接代码:

```
public class NIOEchoClient {

    public static void main(String[] args) {

        int port = 8080;
        new Thread(new NIOClientHandler("127.0.0.1", port)).start();
    }
}
```

NIOClientHandler 客户端处理逻辑代码:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class NIOClientHandler implements Runnable {

    private String host;
    private int port;
    private Selector selector;
    private SocketChannel socketChannel;
    private volatile boolean stop;

    public NIOClientHandler(String host, int port) {
        this.host = host == null ? "127.0.0.1" : host;
        this.port = port;
    }
}
```

```
try {
    selector = Selector.open();
    socketChannel = SocketChannel.open();
    socketChannel.configureBlocking(false);
} catch (IOException e) {
    e.printStackTrace();
    System.exit(1);
}

}

public void run() {

    try {
        doConnect();
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }

    while (!stop) {
        try {
            selector.select(1000);
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> it = selectionKeys.iterator();
            SelectionKey key = null;

            while (it.hasNext()) {
                key = it.next();
                it.remove();

                try {
                    handleInput(key);
                } catch (Exception e) {
```

```

        if (key != null) {
            key.cancel();
            if (key != null) {
                key.channel().close();
            }
        }
    }
} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
}

```

//多路复用器关闭后,所有注册在上面的Channel和Pipe等资源都会被自动注册并关闭,所以不需要重复释放资源

```

    if (selector != null) {
        try {
            selector.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

private void handleInput(SelectionKey key) throws IOException {
    if (key.isValid()) {
        //判断是否连接成功
        SocketChannel sc = (SocketChannel) key.channel();
        if (key.isConnectable()) {
            if (sc.finishConnect()) {
                sc.register(selector, SelectionKey.OP_READ);
                doWrite(sc);
            }
        }
    }
}

```

```
        } else {  
            // 连接失败，退出  
            System.exit(1);  
        }  
    }  
    if (key.isReadable()) {  
        ByteBuffer readBuffer = ByteBuffer.allocate(1024);  
        int readBytes = sc.read(readBuffer);  
        if (readBytes > 0) {  
            readBuffer.flip();  
            byte[] bytes = new byte[readBuffer.remaining()];  
            readBuffer.get(bytes);  
  
            String body = new String(bytes, "UTF-8");  
            System.out.println("Now is :" + body);  
            this.stop = true;  
        } else if (readBytes < 0) {  
            key.cancel();  
            sc.close();  
        } else {  
            ; //读到 0 字节，忽略  
        }  
    }  
}  
}  
  
private void doConnect() throws IOException {  
    //如果直接连接成功，则注册到多路复用器上，发送请求消息，读应答  
    if (socketChannel.connect(new InetSocketAddress(host, port))) {  
        socketChannel.register(selector, SelectionKey.OP_READ);  
        doWrite(socketChannel);  
    } else {  
        socketChannel.register(selector, SelectionKey.OP_CONNECT);  
    }  
}
```

```

    }
}

private void doWrite(SocketChannel sc) throws IOException {
    byte[] req = "Hello,NonBlocking IO.".getBytes();
    ByteBuffer writeBuffer = ByteBuffer.allocate(req.length);
    writeBuffer.put(req);
    writeBuffer.flip();
    sc.write(writeBuffer);
    if (!writeBuffer.hasRemaining()) {
        System.out.println("Send echo content to server succeed!");
    }
}
}

```

6.1.5 NIO2 及 Asynchronous I/O 介绍

JDK7 对原有的 NIO 进行了大幅度的升级，我们称为 NIO2。NIO2 主要改进了 Classic I/O 中 `java.io.File` 类对文件操作的局限性，比如，新引入了 `Path` 及 `Paths`、`Files`、`Directories` 等工具类，支持文件符号链接，扩展了文件的属性支持类型，新的 API 支持文件的复制与移动等。此外，NIO2 还带来了真正意义上的 Asynchronous I/O（异步 I/O），具体实现分为文件 Asynchronous I/O 与网络传输 Asynchronous I/O。

6.1.5.1 NIO2 在 File 操作方面的升级

1. Path 介绍

`Path` 抽象了文件路径，可看作 `java.io.File` 类的升级版。`Path` 是 NIO2 中的基础类之一，NIO2 中大量的 I/O 操作都会用到该类。

下面通过一个代码示例来熟悉该类的使用。

```

import java.io.File;
import java.net.URI;

```

```
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathDemo {

    public static void main(String[] args) {

        //通过工具类 Paths 获取绝对路径 Path
        Path path = Paths.get("C:/java/nio2/2017/BNP.txt");
        System.out.println("path1:" + path);
        path = Paths.get("C:", "java", "nio2", "2017", "test.txt");
        System.out.println("path2:" + path);

        //相对路径 Path
        path = Paths.get("/java/nio2/2017/test.txt");
        System.out.println("path3:" + path);

        //通过 URI 获取 Path
        path = Paths.get(URI.create("file:///java/nio2/2017/test.txt"));
        System.out.println("path4:" + path);

        //通过 FileSystems 获取 Path
        path = FileSystems.getDefault().getPath("/java/nio2/2017",
"test.txt");
        System.out.println("path5:" + path);

        //Path 转成 File
        File path_to_file = path.toFile();
        System.out.println("Path to file name: " + path_to_file.getName());

        //Path 转成 URI
        URI path_to_uri = path.toUri();
        System.out.println("Path to URI: " + path_to_uri);
    }
}
```



```

//获取绝对路径
Path path_to_absolute_path = path.toAbsolutePath();
System.out.println("Path to absolute path: " + path_to_absolute_path.
toString());
}
}

```

运行结果如下:

```

path1:C:/java/nio2/2017/BNP.txt
path2:C:/java/nio2/2017/test.txt
path3:/java/nio2/2017/test.txt
path4:/java/nio2/2017/test.txt
path5:/java/nio2/2017/test.txt
Path to file name: test.txt
Path to URI: file:///java/nio2/2017/test.txt
Path to absolute path: /java/nio2/2017/test.txt

```

2. Paths、Files 工具类介绍

NIO2 在易用性方面很大的一个提升源于提供了一些很好用的工具类, 比如 Paths 和 Files。利用这些工具类, 能够以少量的代码很方便地完成一些常用的操作。

Paths 工具类如下。

◎ `public static Path get(String first, String... more):` 将多个路径段组合成一个新的 Path。

◎ `public static Path get(URI uri):` 根据 uri 创建 Path。

通过以上两个静态方法, 能够很方便地构造 Path 对象。

Files 工具类常用方法如下。

◎ `public static InputStream newInputStream(Path path, OpenOption... options):` 获得文

件输入流。

- ◎ `public static OutputStream newOutputStream(Path path, OpenOption... options)`: 获得文件输出流。
- ◎ `public static SeekableByteChannel newByteChannel(Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs)`: 获得文件通道。
- ◎ `public static DirectoryStream<Path> newDirectoryStream(Path dir)`: 能够通过返回对象 `DirectoryStream` 获取迭代器 `Iterator`, 进而获取到参数目录 `dir` 下的一级目录或者文件。
- ◎ `public static Path createFile(Path path, FileAttribute<?>... attrs)`: 创建文件, 并返回所创建文件的 `Path` 对象。
- ◎ `public static Path createDirectory(Path dir, FileAttribute<?>... attrs)`: 创建目录, 并返回所创建目录的 `Path` 对象。
- ◎ `public static Path createDirectories(Path dir, FileAttribute<?>... attrs)`: 递归创建目录, 并返回所创建目录的 `Path` 对象。
- ◎ `public static Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)`: 创建临时文件。
- ◎ `public static Path createSymbolicLink(Path link, Path target, FileAttribute<?>... attrs)`: 创建指定 `Path` 的符号链接。
- ◎ `public static void delete(Path path)`: 删除 `Path` 所代表的文件。
- ◎ `public static boolean deleteIfExists(Path path)`: 如果 `Path` 所代表的文件存在, 删除 `Path` 所代表的文件。
- ◎ `public static Path copy(Path source, Path target, CopyOption... options)`: 文件复制。
- ◎ `public static Path move(Path source, Path target, CopyOption... options)`: 移动文件。

- ◎ `public static Path readSymbolicLink(Path link)`: 读取符号链接文件。
- ◎ `public static boolean isSameFile(Path path, Path path2)`: 判断是否是同一个文件。
- ◎ `public static boolean isReadable(Path path)`: 判断文件是否可读。
- ◎ `public static boolean isWritable(Path path)`: 判断文件是否可写。
- ◎ `public static boolean isExecutable(Path path)`: 判断文件是否可执行。
- ◎ `public static BufferedReader newBufferedReader(Path path, Charset cs)`: 获取文件的缓冲字节输入流。
- ◎ `public static BufferedWriter newBufferedWriter(Path path, Charset cs, OpenOption... options)`: 获取文件的缓冲字节输出流。
- ◎ `public static long copy(InputStream in, Path target, CopyOption... options)`: 从文件输入流获取文件内容实现文件复制。
- ◎ `public static long copy(Path source, OutputStream out)`: 将文件复制到文件输出流。
- ◎ `public static byte[] readAllBytes(Path path)`: 读取文件, 返回文件内容字节数组。
- ◎ `public static List<String> readAllLines(Path path, Charset cs)`: 读取文件, 返回文件内容字符串数组。
- ◎ `public static Path write(Path path, byte[] bytes, OpenOption... options)`: 将字节数组内容写入指定的文件。
- ◎ `public static Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)`: 将字符串数组写入指定的文件。

基本上文件的常用操作都可以通过以上的静态方法完成, 下面给出部分方法的代码示例。

```
import java.io.BufferedReader;  
import java.io.BufferedWriter;
```

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.*;
import java.util.Iterator;

public class FilesDemo {

    public static void main(String[] args) throws IOException {
        //遍历 linux 根目录/下的子目录
        DirectoryStream<Path> directoryStream = Files.newDirectoryStream
(Paths.get("/"));
        Iterator<Path> iterator = directoryStream.iterator();
        while (iterator.hasNext()) {
            Path path = iterator.next();
            System.out.println(path);
        }

        //递归创建文件目录
        Path path = Files.createDirectories(Paths.get("/Users/liyebing/
test/test/"));

        System.out.println(path.getFileName());

        //创建文件 test.txt
        Path file = Files.createFile(Paths.get("/Users/liyebing/test/test/
test.txt"));

        //使用缓冲字符流写入文件内容
        Charset charset = Charset.forName("UTF-8");
        String text = "Hello,java NIO2!";
        try {
            BufferedWriter writer = Files.newBufferedWriter(file, charset,
StandardOpenOption.APPEND);
            writer.write(text);
            writer.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

```

    }

    //使用缓冲字符流读取文件内容
    BufferedReader reader = null;
    try {
        reader = Files.newBufferedReader(file, charset);
        String line = null;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        System.err.println(e);
    } finally {
        if (reader != null) {
            reader.close();
        }
    }
}
}

```

3. WatchService 接口

java.nio.file.WatchService 接口的引入是 NIO2 对于文件操作功能的一个很大的提升，它提供了通过应用程序监听操作系统文件变更事件的能力。通过在 Path 上注册所需要监听的事件（文件创建、文件修改、文件删除），一旦被监听的文件发生变更，应用程序能够实时感知到这种变化。

下面通过一个代码示例来学习这种新特性。

```

public class WatchServiceDemo {

    public void watchDir(Path path) throws IOException, InterruptedException
    {
        try {

```

```
//创建 WatchService 实例
WatchService watchService =
FileSystems.getDefault().newWatchService();

//注册 WatchService 所监听的事件（目录创建、目录修改、目录删除）
path.register(watchService,
StandardWatchEventKinds.ENTRY_CREATE, StandardWatchEventKinds.ENTRY_MODIFY,
StandardWatchEventKinds.ENTRY_DELETE);

//无限循环获取监听到的事件
while (true) {
    final WatchKey key = watchService.take();
    for (WatchEvent<?> watchEvent : key.pollEvents()) {
        final Kind<?> kind = watchEvent.kind();
        //忽略 OVERFLOW 事件
        if (kind == StandardWatchEventKinds.OVERFLOW) {
            continue;
        }

        final WatchEvent<Path> watchEventPath = (WatchEvent<Path>)
watchEvent;

        final Path fileName = watchEventPath.context();
        //打印事件类型及发生事件的文件名称
        System.out.println(kind + " : " + fileName);
    }
    //重置 Key
    boolean valid = key.reset();
    //如果 Key 无效（比如监听的文件被删除），则退出
    if (!valid) {
        break;
    }
}
} catch (IOException e) {
    e.printStackTrace();
}
```

```

    }
}

public static void main(String[] args) {
    final Path path = Paths.get("D://");
    WatchServiceDemo watch = new WatchServiceDemo();
    try {
        watch.watchDir(path);
    } catch (IOException | InterruptedException ex) {
        System.err.println(ex);
    }
}
}

```

通过以上代码，一旦目录 D:// 下面的文件发生变更，将实时打印出文件变更的事件及发生变更的文件名称。

6.1.5.2 文件读写 Asynchronous I/O

NIO2 通过 `AsynchronousFileChannel` 提供了异步读写取文件的功能。通过 `AsynchronousFileChannel` 异步读写文件有 `CompletionHandler` 与 `Future` 两种方式。下面分别通过相应的代码示例来说明如何使用这两种方式来异步读写文件^①。

演示使用 `CompletionHandler` 异步读取文件内容：

```

//打开文件通道，获取异步 Channel 对象
AsynchronousFileChannel ch = AsynchronousFileChannel.open(Paths.get(
    "/Users/liyebing/p.txt"), StandardOpenOption.READ);

//读取文件临时 ByteBuffer 对象
ByteBuffer buffer = ByteBuffer.allocate(1024);

//文件读取初始位置

```

① 具体参考：<http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html>, <http://ivan4126.blog.163.com/blog/static/209491092201361621344523/>。

```

        long position = 0;

        final List<Integer> byteReadResultHolder = new
ArrayList<Integer>(1);

        //文件读取结果内容
        List<Byte> totalByteList = new ArrayList<Byte>();
        while (true) {
            //每次开始读之前，先重置读取结果计数器
            byteReadResultHolder.clear();

            //异步读文件
            final CountDownLatch latch = new CountDownLatch(1);
            ch.read(buffer, position, null, new CompletionHandler<Integer,
Void>() {

                public void completed(Integer result, Void attachment) {
                    //保存本次读取的结果，若 result == -1，则代表数据已经读完
                    byteReadResultHolder.add(result);
                    latch.countDown();
                }

                public void failed(Throwable exc, Void attachment) {
                    //读文件失败，打印失败结果
                    System.out.println("read failure:" + exc.toString());
                    latch.countDown();
                }

            });

            //等待异步文件读取操作完成
            latch.await();

            //如果读取完毕，则直接退出
            if (byteReadResultHolder.size() <= 0 ||
(byteReadResultHolder.size() > 0 && byteReadResultHolder.get(0) == -1)) {
                break;
            }
        }
    }
}

```



```

    }

    //读取本次从文件读取到 Buffer 里面的数据
    buffer.flip();
    //计算下次文件读取的开始位置
    position = position + buffer.limit();
    while (buffer.hasRemaining()) {
        byte[] data = new byte[buffer.limit()];
        buffer.get(data);
        //将本次从文件里面读取到的数据保存到总的读取结果 byte 数组
totalByteList
        for (byte by : data) {
            totalByteList.add(by);
        }
    }

    //重置临时读取结果 ByteBuffer 对象，以便下次循环使用
    buffer.clear();
}

//将读取结果 byte 数组 totalByteList 转换成字符串并打印出来
byte[] bytes = new byte[totalByteList.size()];
for (int i = 0; i < totalByteList.size(); i++) {
    bytes[i] = totalByteList.get(i);
}
String fileContent = new String(bytes);
System.out.println("content :" + fileContent);
ch.close();

```

演示使用 CompletionHandler 异步写文件：

```

//若文件 p.txt 不存在，则创建该文件
Path path = Paths.get("/Users/liyebing/p.txt");
if (!Files.exists(path)) {
    Files.createFile(path);
}

```

```
    }  
    //获得异步写入文件的 fileChannel  
    AsynchronousFileChannel fileChannel = AsynchronousFileChannel.  
open(path, StandardOpenOption.WRITE);  
  
    //指定写入文件的开始位置  
    long position = 0;  
    //构造文件写入内容  
    ByteBuffer buffer = ByteBuffer.allocate(1024);  
    buffer.put("你好,java Blocking I/O!".getBytes());  
    buffer.flip();  
  
    //异步写入操作  
    final CountdownLatch latch = new CountdownLatch(1);  
    fileChannel.write(buffer, position, buffer, new  
CompletionHandler<Integer, ByteBuffer>() {  
        public void completed(Integer result, ByteBuffer attachment) {  
            //写入操作成功完成  
            latch.countDown();  
        }  
  
        public void failed(Throwable exc, ByteBuffer attachment) {  
            //写入操作出错  
            latch.countDown();  
            exc.printStackTrace();  
        }  
    });  
  
    //等待异步写入完成  
    latch.await();  
    //写入操作完成后打印信息到控制台  
    System.out.println("Async Write File done");
```

演示使用 Future 异步读文件：

```
//打开文件通道，获取异步 Channel 对象
AsynchronousFileChannel channel = AsynchronousFileChannel.open(Paths.get
("/Users/liyebing/p.txt"), StandardOpenOption.READ);
//读取文件临时 ByteBuffer 对象
ByteBuffer buffer = ByteBuffer.allocate(1024);
//文件读取位置
long position = 0;
//文件读取结果内容
List<Byte> totalByteList = new ArrayList<Byte>();

while (true) {
    //使用 Future 异步读取文件内容
    Future<Integer> operation = channel.read(buffer, position);
    //do nothing，等待读取完成
    while (!operation.isDone()) ;

    buffer.flip();
    //若本次没有读取到数据，说明已经读取完毕，直接跳出循环
    if (!buffer.hasRemaining()) {
        break;
    }

    //计算下次文件读取的开始位置
    position = position + buffer.limit();
    while (buffer.hasRemaining()) {
        byte[] data = new byte[buffer.limit()];
        buffer.get(data);
        //将本次从文件里面读取到的数据保存到总的读取结果 byte 数组
        totalByteList
        for (byte by : data) {
            totalByteList.add(by);
        }
    }
}
```

```
    }  
    //重置临时读取结果 ByteBuffer 对象，以便下次循环使用  
    buffer.clear();  
}  
  
//将读取结果 byte 数组 totalByteList 转换成字符串并打印出来  
byte[] bytes = new byte[totalByteList.size()];  
for (int i = 0; i < totalByteList.size(); i++) {  
    bytes[i] = totalByteList.get(i);  
}  
String fileContent = new String(bytes);  
System.out.println("content :" + fileContent);  
channel.close();
```

演示使用 Future 异步写文件：

```
//若文件 p.txt 不存在，则创建该文件  
Path path = Paths.get("/Users/liyebing/p.txt");  
if (!Files.exists(path)) {  
    Files.createFile(path);  
}  
  
//获得文件异步写入 fileChannel  
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open  
(path, StandardOpenOption.WRITE);  
  
//指定写入文件的开始位置  
long position = 0;  
  
//构造写入文件的内容  
ByteBuffer buffer = ByteBuffer.allocate(1024);  
buffer.put("你好,java Blocking I/O!".getBytes());  
buffer.flip();  
  
//写入文件操作
```

```

Future<Integer> operation = fileChannel.write(buffer, position);
buffer.clear();

//等待写入操作完成, 关闭文件通道
while (!operation.isDone()) ;
fileChannel.close();

//写入操作完成后打印信息到控制台
System.out.println("Async Write File done");

```

6.1.5.3 Java 异步 I/O 网络通信实现

NIO2 通过引入 `AsynchronousSocketChannel` 与 `AsynchronousServerSocketChannel` 实现了异步 I/O 网络通信模型。

下面通过实现一个网络回显服务来演示相应 API 的使用。

`AIOEchoServer` 启动服务端代码:

```

public class AIOEchoServer {

    public static void main(String[] args) {
        int port = 8080;
        AsyncEchoServerHandler timeServer = new AsyncEchoServerHandler
(port);
        new Thread(timeServer).start();
    }
}

```

`AsyncEchoServerHandler` 服务端实现代码:

```

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.util.concurrent.CountDownLatch;

```

```
public class AsyncEchoServerHandler implements Runnable {

    private int port;

    CountdownLatch latch;

    AsynchronousServerSocketChannel asynchronousServerSocketChannel;

    public AsyncEchoServerHandler(int port) {
        this.port = port;
        try {
            //获取 AsynchronousServerSocketChannel 对象
            asynchronousServerSocketChannel
AsynchronousServerSocketChannel.open();
            //绑定服务端口
            asynchronousServerSocketChannel.bind(new
InetSocketAddress(port));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void run() {
        latch = new CountdownLatch(1);
        doAccept();
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

public void doAccept() {
    //接受客户端连接
    asynchronousServerSocketChannel.accept(this,
                                           new
AcceptCompletionHandler());
}
}

```

AcceptCompletionHandler 服务端接入客户端请求代码:

```

import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;

public class AcceptCompletionHandler implements CompletionHandler
<AsynchronousSocketChannel, AsyncEchoServerHandler> {

    public void completed(AsynchronousSocketChannel result,
AsyncEchoServerHandler attachment) {
        //循环接入客户端
        attachment.asynchronousServerSocketChannel.accept(attachment,
this);

        ByteBuffer buffer = ByteBuffer.allocate(1024);
        result.read(buffer, buffer, new ReadCompletionHandler(result));
    }

    public void failed(Throwable exc, AsyncEchoServerHandler attachment) {
        exc.printStackTrace();
        attachment.latch.countDown();
    }
}

```

ReadCompletionHandler 服务端业务逻辑实现代码:

```

import java.nio.ByteBuffer;

```

```
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;

public class ReadCompletionHandler implements CompletionHandler<Integer,
ByteBuffer> {
    private AsynchronousSocketChannel channel;

    public ReadCompletionHandler(AsynchronousSocketChannel channel) {
        if (this.channel == null) {
            this.channel = channel;
        }
    }

    public void completed(Integer result, ByteBuffer attachment) {
        //获取客户端传入的数据
        attachment.flip();
        byte[] body = new byte[attachment.remaining()];
        attachment.get(body);

        try {
            //将客户端传入的数据打印到控制台
            String req = new String(body, "UTF-8");
            System.out.println("echo content:" + req);
            //将接收的数据回传给客户端
            doWrite(req);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void doWrite(String currentTime) {
        if (currentTime != null && currentTime.trim().length() > 0) {
            byte[] bytes = currentTime.getBytes();
        }
    }
}
```



```

        final ByteBuffer writeBuffer = ByteBuffer.allocate
(bytes.length);

        writeBuffer.put(bytes);
        writeBuffer.flip();

        channel.write(writeBuffer, writeBuffer, new CompletionHandler
<Integer, ByteBuffer>() {
            public void completed(Integer result, ByteBuffer buffer) {
                //如果没有发送完成, 继续发送
                if (buffer.hasRemaining()) {
                    channel.write(buffer, buffer, this);
                }
            }

            public void failed(Throwable exc, ByteBuffer buffer) {
                try {
                    channel.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

    public void failed(Throwable exc, ByteBuffer attachment) {
        try {
            channel.close();
        } catch (Exception e) {
        }
    }
}

```

AIOEchoClient 客户端启动代码：

```
public class AIOEchoClient {

    public static void main(String[] args) {
        int port = 8080;
        new Thread(new AsyncEchoClientHandler("127.0.0.1", port)).start();
    }
}
```

AsyncEchoClientHandler 客户端业务逻辑实现：

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.CountDownLatch;

public class AsyncEchoClientHandler implements CompletionHandler<Void,
AsyncEchoClientHandler>, Runnable {

    private AsynchronousSocketChannel client;
    private String host;
    private int port;
    private CountDownLatch latch;

    public AsyncEchoClientHandler(String host, int port) {
        this.host = host;
        this.port = port;
        try {
            client = AsynchronousSocketChannel.open();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

}
```

```

public void run() {
    latch = new CountDownLatch(1);
    client.connect(new InetSocketAddress(host, port), this, this);
    try {
        latch.await();
        client.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

```

public void completed(Void result, AsyncEchoClientHandler attachment)
{
    //准备写入服务端的数据
    byte[] req = "你好,java Asynchronous IO.".getBytes();
    ByteBuffer writeBuffer = ByteBuffer.allocate(req.length);
    writeBuffer.put(req);
    writeBuffer.flip();

    //将数据写入服务端
    client.write(writeBuffer, writeBuffer, new CompletionHandler
<Integer, ByteBuffer>() {
        //客户端数据写入服务端写入完成
        public void completed(Integer result, ByteBuffer buffer) {
            if (buffer.hasRemaining()) {
                client.write(buffer, buffer, this);
            } else {
                ByteBuffer readBuffer = ByteBuffer.allocate(1024);
                //读取从服务端传回的数据
                client.read(readBuffer, readBuffer, new CompletionHandler
```

```
<Integer, ByteBuffer>() {  
    //服务端数据成功传回处理逻辑  
    public void completed(Integer result, ByteBuffer buffer)  
{  
        buffer.flip();  
        byte[] bytes = new byte[buffer.remaining()];  
        buffer.get(bytes);  
  
        String body;  
        try {  
            //将服务端传回的数据打印到控制台  
            body = new String(bytes, "UTF-8");  
            System.out.println("echo content is:" + body);  
            latch.countDown();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    //服务端数据返回出错  
    public void failed(Throwable exc, ByteBuffer buffer) {  
        try {  
            client.close();  
            latch.countDown();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
    }  
});  
}  
  
//客户端数据写入服务端写入出错
```

```
public void failed(Throwable exc, ByteBuffer attachment) {  
    try {  
        client.close();  
        latch.countDown();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
});  
}  
  
public void failed(Throwable exc, AsyncEchoClientHandler attachment) {  
    try {  
        client.close();  
        latch.countDown();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
}
```

6.2 Netty 使用介绍

Netty 是著名的 NIO 开源框架，提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。其主要开发人员 Trustin Lee 也是另一个著名 NIO 框架 Mina 的主创人员。Netty 晚于 Mina 开发，对比 Mina，Netty 在设计及易用性上都有较大改进。本节将介绍 Netty 的使用方法。

6.2.1 Netty 开发入门

Netty 目前的最新版本是 netty-4.1.8.Final.tar.bz2, 参考 <http://netty.io/downloads.html>。Netty5 短暂地存在过一段时间, 但目前已经废弃, 废弃原因参考 <https://github.com/netty/netty/issues/4466>。

我们本次使用 Netty 的最新版本, 其 Maven 依赖如下:

```
<dependency>
<groupId>io.netty</groupId>
<artifactId>netty-all</artifactId>
<version>4.1.8.Final</version>
</dependency>
```

还是以网络回显应用为例, 通过具体的代码示例来学习 Netty 的一般使用流程。

EchoServer 启动 Netty 服务端:

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
```

```
public class EchoServer {

    public static void main(String[] args) throws Exception {
        int port = 8080;
        new EchoServer().bind(port);
    }
}
```

```

public void bind(int port) throws Exception {
    //创建两个 EventLoopGroup 实例
    //EventLoopGroup 是包含一组专门用于处理网络事件的 NIO 线程组
    EventLoopGroup bossGroup = new NioEventLoopGroup();
    EventLoopGroup workerGroup = new NioEventLoopGroup();

    try {
        //创建服务端辅助启动类 ServerBootstrap 对象
        ServerBootstrap b = new ServerBootstrap();
        //设置 NIO 线程组
        b.group(bossGroup, workerGroup)
            // 设置 NioServerSocketChannel , 对应于 JDK NIO 类
            //ServerSocketChannel
            .channel(NioServerSocketChannel.class)
            //设置 TCP 参数, 连接请求的最大队列长度
            .option(ChannelOption.SO_BACKLOG, 1024)
            //设置 I/O 事件处理类, 用来处理消息的编解码及我们的业务逻辑
            .childHandler(new ChannelInitializer<NioSocketChannel>()
            {
                @Override
                protected void initChannel(NioSocketChannel ch) throws
                Exception {
                    ch.pipeline().addLast(new EchoServerHandler());
                }
            });

        //绑定端口, 同步等待成功
        ChannelFuture f = b.bind(port).sync();
        //等待服务端监听端口关闭
        f.channel().closeFuture().sync();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {

```

```
        //优雅退出，释放线程池资源
        bossGroup.shutdownGracefully();
        workerGroup.shutdownGracefully();
    }
}
}
```

EchoServerHandler 服务端 I/O 事件处理类：

```
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;

public class EchoServerHandler extends SimpleChannelInboundHandler {

    @Override
    public void channelRead0(ChannelHandlerContext ctx, Object msg) throws
Exception {
        //接收客户端发来的数据，使用 buf.readableBytes() 获取数据大小，并转换成 byte
数组

        ByteBuf buf = (ByteBuf) msg;
        byte[] req = new byte[buf.readableBytes()];
        buf.readBytes(req);

        //将 byte 数组转成字符串，在控制台打印输出
        String body = new String(req, "UTF-8");
        System.out.println("receive data from client:" + body);

        //将接收到的客户端发来的数据回写到客户端
        ByteBuf resp = Unpooled.copiedBuffer(body.getBytes());
        ctx.write(resp);
    }
}
```



```

//发生异常，关闭链路
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    ctx.close();
}

//将发送缓冲区中的消息全部写入 SocketChannel 中
@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
    ctx.flush();
}
}

```

至此网络回显应用 Netty 服务端代码开发完成。

EchoClient 启动 Netty 客户端：

```

import io.netty.bootstrap.Bootstrap;
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;

public class EchoClient {

    public static void main(String[] args) throws Exception {
        int port = 8080;

```

```

        new EchoClient().connect(port, "127.0.0.1");
    }

    public void connect(int port, String host) throws Exception {
        //创建客户端处理 I/O 读写的 NIO 线程组
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            //创建客户端辅助启动类
            Bootstrap b = new Bootstrap();
            //设置 NIO 线程组
            b.group(group)

                //设置 NioSocketChannel, 对应于 JDK NIO 类 SocketChannel 类
                .channel(NioSocketChannel.class)

                //设置 TCP 参数 TCP_NODELAY
                .option(ChannelOption.TCP_NODELAY, true)

                .handler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) throws
Exception {

                        //配置客户端处理网络 I/O 事件的类
                        ch.pipeline().addLast(new EchoClientHandler());
                    }
                });

            //发起异步连接操作
            ChannelFuture f = b.connect(host, port).sync();

            //构造客户端发送的数据 ByteBuf 对象
            byte[] req = "你好, Netty!".getBytes();
            ByteBuf messageBuffer = Unpooled.buffer(req.length);
            messageBuffer.writeBytes(req);

            //向服务端发送数据
            ChannelFuture channelFuture =

```

```
f.channel().writeAndFlush(messageBuffer);
        channelFuture.syncUninterruptibly();

        //等待客户端链路关闭
        f.channel().closeFuture().sync();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        //优雅退出，释放NIO线程组
        group.shutdownGracefully();
    }
}
}
```

EchoClientHandler 处理客户端 I/O 事件类:

```
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import java.util.logging.Logger;

public class EchoClientHandler extends SimpleChannelInboundHandler {
    private static final Logger logger = Logger.getLogger(
        EchoClientHandler.class.getName());

    //服务端响应请求返回数据的时候会自动调用该方法，我们通过实现该方法来接收服务端返回
    的数据，并实现客户端调用的业务逻辑
    @Override
    public void channelRead0(ChannelHandlerContext ctx, Object msg) throws
    Exception {
        //获取服务端返回的数据 buf
        ByteBuf buf = (ByteBuf) msg;
        byte[] req = new byte[buf.readableBytes()];
        buf.readBytes(req);
```

```
//将服务端返回的 byte 数组转换成字符串，并打印到控制台
String body = new String(req, "UTF-8");
System.out.println("receive data from server :" + body);
}

//发生异常，关闭链路
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    logger.warning("Unexpected exception from downstream : " +
cause.getMessage());
    ctx.close();
}
}
```

先运行 EchoServer 类的 main 方法，将 Netty 服务启动。再运行 EchoClient 类的 main 方法，发起 Netty 客户端调用。运行结果如下：

```
receive data from server :你好,Netty!
```

对照之前 NIO 实现网络回显服务的代码，可以发现 Netty 简化了 NIO 繁杂的 API 调用。这也是我们使用 Netty 而不直接使用 NIO 进行网络应用开发的原因之一。此外 NIO 自身也有 Bug，例如导致 CPU 100% 的 Bug，参考 <http://blog.csdn.net/chenxuegui1234/article/details/17767433>，而 Netty 较好地规避了这些已知 Bug。

以上代码只是进行了一次调用，下面我们测试一下，多次循环调用 Netty 服务端的返回结果。我们对 EchoClient 中调用 Netty 的部分代码做如下改造：

```
.....省略相同的重复代码.....
//发起异步连接操作
ChannelFuture f = b.connect(host, port).sync();

for (int i = 0; i < 1000; i++) {
```

```
//构造客户端发送的数据 ByteBuffer 对象
byte[] req = "你好,Netty!".getBytes();
ByteBuffer messageBuffer = Unpooled.buffer(req.length);
messageBuffer.writeBytes(req);

//向服务端发送数据
ChannelFuture channelFuture = f.channel().writeAndFlush(messageBuffer);
channelFuture.syncUninterruptibly();
}

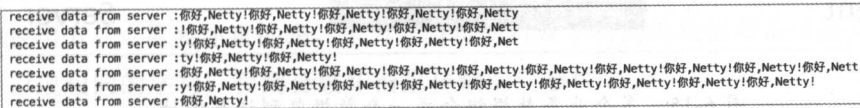
//等待客户端链路关闭
f.channel().closeFuture().sync();

.....省略相同的重复代码.....
```

运行改造之后的代码，发现运行结果中并没有按照预期逐行打印如下字符串：

```
receive data from server :你好,Netty!
```

而是出现如下打印结果字符串片段，如图 6-13 所示。



```
receive data from server :你好,Netty!你好,Netty!你好,Netty!你好,Netty!你好,Netty!
receive data from server :!你好,Netty!你好,Netty!你好,Netty!你好,Netty!你好,Nett
receive data from server :y!你好,Netty!你好,Netty!你好,Netty!你好,Nett
receive data from server :你好,Netty!你好,Netty!你好,Netty!你好,Netty!你好,Nett
receive data from server :y!你好,Netty!你好,Netty!你好,Netty!你好,Netty!你好,Nett
receive data from server :你好,Netty!
```

图 6-13 字符串片段

这就是 TCP 传输过程中出现的粘包/半包问题导致的现象。

为什么会出现粘包/半包的现象呢？

- ◎ 应用程序写入的数据大于套接字缓冲区大小，这将会导致半包现象。
- ◎ 应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包现象。
- ◎ 当 TCP 报文长度减去 TCP 头部长度大于 MSS（最大报文长度）的时候将发生半包。

◎ 接收方法未能及时读取套接字缓冲区数据，将发生粘包。

大家可以想象，TCP 传输数据按照字节包数据流的形式进行传输，就像流水一般，连在一起，TCP 底层无法获知业务数据的具体含义，无法按照业务含义进行分包，只会按照 TCP 缓冲区的实际情况进行包的划分。业务数据被分拆为多个数据包，这些数据包到达目的地有以下三种情况。

◎ 按照业务数据本身的边界逐个到达目的地，如图 6-14 所示。

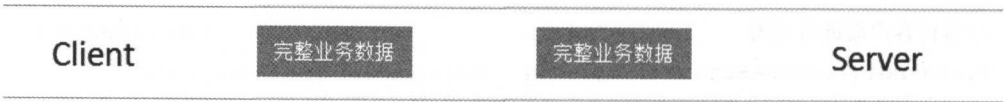


图 6-14 按照业务数据边界逐个到达目的地

◎ 多个业务数据组合成一个数据包到达目的地，这种情况即为粘包的情况，如图 6-15 所示。

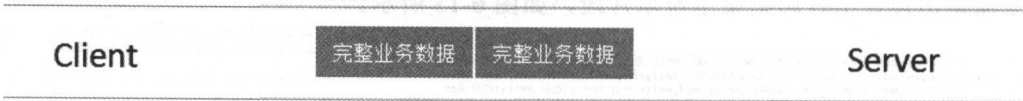


图 6-15 多个业务数据组合成一个数据包到达目的地

◎ 到达目的地的数据包中只包含部分业务数据，这种情况即为半包的情况，如图 6-16 所示。

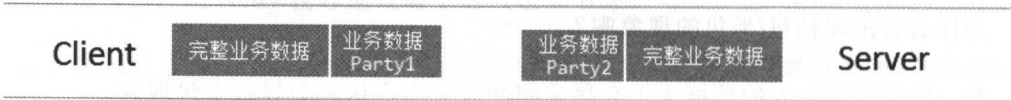


图 6-16 数据包中包含业务数据的某一部分

Netty 传输数据我们一般采用的是 TCP/IP 协议，也会出现上述粘包/半包问题，下一节将详细介绍如何解决 Netty 使用过程中的粘包/半包问题。

6.2.2 Netty 粘包/半包问题解决

解决粘包/半包问题的本质是能够区分完整的业务应用数据边界，能够按照边界完整地接受 Netty 传输的数据。

6.2.2.1 字符串类型的消息的编解码

对于字符串类型消息，可以使用 Netty 提供的编码解码工具类解决粘包/半包问题，具体有如下几种工具类组合。

- ◎ `DelimiterBasedFrameDecoder+StringDecoder` 利用特殊分隔符作为消息的结束标志。
- ◎ `LineBasedFrameDecoder+StringDecoder` 组合以换行符作为消息的结束标志。
- ◎ `FixedLengthFrameDecoder+StringDecoder` 按照固定长度获取消息。

1. `DelimiterBasedFrameDecoder` 与 `StringDecoder` 组合

`DelimiterBasedFrameDecoder` 的原理很简单，就是使用特殊的字符，将其作为数据包的结束位置。下面通过相应的代码示例来说明 `DelimiterBasedFrameDecoder+StringDecoder` 组合的使用方法。需要注意的是，作为分隔符的特殊字符的选取需要保证正式报文数据里面不会出现相同的字符，否则会破坏消息传输的完整性。

Netty 服务端 `DelimiterBaseServer` 的实现：

```
public class DelimiterBaseServer {

    //字符串分隔符
    private static final String delimiter_tag = "@#";

    public static void main(String[] args) throws Exception {
        int port = 8080;
        //绑定服务端口，并启动 Netty 服务端
        new DelimiterBaseServer().bind(port);
    }
}
```

```
public void bind(int port) throws Exception {
    EventLoopGroup bossGroup = new NioEventLoopGroup();
    EventLoopGroup workerGroup = new NioEventLoopGroup();

    try {
        ServerBootstrap b = new ServerBootstrap();
        b.group(bossGroup, workerGroup)
            .channel(NioServerSocketChannel.class)
            .option(ChannelOption.SO_BACKLOG, 1024)
            .childHandler(new ChannelInitializer<NioSocketChannel>()
            {
                @Override
                protected void initChannel(NioSocketChannel ch) throws
Exception {
                    //设置 DelimiterBasedFrameDecoder 处理器
                    ByteBuf delimiter = Unpooled.copiedBuffer
(delimiter_tag.getBytes());
                    ch.pipeline().addLast(new DelimiterBasedFrame
Decoder(1024, delimiter));
                    //设置 StringDecoder 处理器
                    ch.pipeline().addLast(new StringDecoder());
                    ch.pipeline().addLast(new
DelimiterBaseServerHandler());
                }
            });

        ChannelFuture f = b.bind(port).sync();
        f.channel().closeFuture().sync();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        bossGroup.shutdownGracefully();
        workerGroup.shutdownGracefully();
    }
}
```



```

    }
}

```

服务端业务逻辑处理类负责接收客户端发送的数据，并回写给客户端，服务端业务逻辑处理类 `DelimiterBaseServerHandler` 实现如下：

```

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import java.util.concurrent.atomic.AtomicInteger;

public class DelimiterBaseServerHandler extends
SimpleChannelInboundHandler {

    //字符串分隔符
    private static final String delimiter_tag = "@#";
    //计数器
    private static final AtomicInteger counter = new AtomicInteger(0);

    @Override
    public void channelRead0(ChannelHandlerContext ctx, Object msg) throws
Exception {
        //接收客户端发送的字符串，并打印到控制台
        String content = (String) msg;
        System.out.println("received from client:" + content + " counter:"
+ counter.addAndGet(1));

        //加入分隔符，将数据重新发送到客户端
        content += delimiter_tag;
        ByteBuf echo = Unpooled.copiedBuffer(content.getBytes());
        ctx.writeAndFlush(echo);
    }
}

```

```
@Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        ctx.close();
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
        ctx.flush();
    }
}
```

Netty 客户端 `DelimiterBaseClient` 发起调用，循环写入服务端 1000 条字符串。实现代码如下：

```
import io.netty.bootstrap.Bootstrap;
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.DelimiterBasedFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;

public class DelimiterBaseClient {
    //字符串分隔符
    private static final String delimiter_tag = "@#";

    public static void main(String[] args) throws Exception {
        int port = 8080;
```

```

        new DelimiterBaseClient().connect(port, "127.0.0.1");
    }

    public void connect(int port, String host) throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();
            b.group(group)
              .channel(NioSocketChannel.class)
              .option(ChannelOption.TCP_NODELAY, true)
              .handler(new ChannelInitializer<NioSocketChannel>() {
                  @Override
                  protected void initChannel(NioSocketChannel ch) throws
Exception {

                        //设置 DelimiterBasedFrameDecoder 处理器
                        ByteBuf delimiter =
Unpooled.copiedBuffer(delimiter_tag.getBytes());
                        ch.pipeline().addLast(new
DelimiterBasedFrameDecoder(1024, delimiter));

                        //设置 StringDecoder 处理器
                        ch.pipeline().addLast(new StringDecoder());
                        //配置客户端处理网络 I/O 事件的类
                        ch.pipeline().addLast(new
DelimiterBaseClientHandler());
                    }
                });

            //发起异步连接操作
            ChannelFuture f = b.connect(host, port).sync();

            //循环发送 1000 次
            for (int i = 0; i < 1000; i++) {
                //构造客户端发送的数据 ByteBuf 对象

```

```

        String content = "你好,Netty!" + delimiter_tag;
        byte[] req = content.getBytes();
        ByteBuf messageBuffer = Unpooled.buffer(req.length);
        messageBuffer.writeBytes(req);

        //向服务端发送数据
        ChannelFuture channelFuture =
f.channel().writeAndFlush(messageBuffer);
        channelFuture.syncUninterruptibly();
    }
    f.channel().closeFuture().sync();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    group.shutdownGracefully();
}
}
}

```

客户端业务处理 **DelimiterBaseClientHandler**，负责接收服务端回写的数据，并打印控制台。具体实现如下：

```

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Logger;

public class DelimiterBaseClientHandler extends SimpleChannelInbound
Handler {

    private static final Logger logger = Logger.getLogger
(DelimiterBaseClientHandler.class.getName());

    //计数器
    private static final AtomicInteger counter = new AtomicInteger(0);

```

```

@Override
public void channelRead0(ChannelHandlerContext ctx, Object msg) throws
Exception {
    //获取服务端返回的数据，并打印到控制台
    String content = (String) msg;
    System.out.println("received from server:" + content + " counter:"
+ counter.addAndGet(1));
}

//发生异常，关闭链路
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    logger.warning("Unexpected exception from downstream : " +
cause.getMessage());
    ctx.close();
}
}

```

先运行 `DelimiterBaseServer` 中的 `main` 方法，将 Netty 服务启动，再运行 `DelimiterBaseClient` 中的 `main` 方法，循环发送 1000 条字符串给服务端，运行结果如下。

服务端控制台打印内容如下：

```

received from client:你好,Netty! counter:1
received from client:你好,Netty! counter:2
received from client:你好,Netty! counter:3
received from client:你好,Netty! counter:4
received from client:你好,Netty! counter:5
received from client:你好,Netty! counter:6
received from client:你好,Netty! counter:7
received from client:你好,Netty! counter:8
received from client:你好,Netty! counter:9

```

```
.....省略.....
.....省略.....
received from client:你好,Netty! counter:989
received from client:你好,Netty! counter:990
received from client:你好,Netty! counter:991
received from client:你好,Netty! counter:992
received from client:你好,Netty! counter:993
received from client:你好,Netty! counter:994
received from client:你好,Netty! counter:995
received from client:你好,Netty! counter:996
received from client:你好,Netty! counter:997
received from client:你好,Netty! counter:998
received from client:你好,Netty! counter:999
received from client:你好,Netty! counter:1000
```

客户端控制台打印内容如下:

```
received from server:你好,Netty! counter:1
received from server:你好,Netty! counter:2
received from server:你好,Netty! counter:3
received from server:你好,Netty! counter:4
received from server:你好,Netty! counter:5
received from server:你好,Netty! counter:6
received from server:你好,Netty! counter:7
received from server:你好,Netty! counter:8
received from server:你好,Netty! counter:9
.....省略.....
.....省略.....
received from server:你好,Netty! counter:992
received from server:你好,Netty! counter:993
received from server:你好,Netty! counter:994
received from server:你好,Netty! counter:995
received from server:你好,Netty! counter:996
received from server:你好,Netty! counter:997
```

```
received from server:你好,Netty! counter:998
received from server:你好,Netty! counter:999
received from server:你好,Netty! counter:1000
```

可以发现,无论是客户端还是服务端的控制台都完整地打印了 1000 条字符串,说明通过 `DelimiterBasedFrameDecoder+StringDecoder` 组合已经解决了半包/粘包问题。

2. `LineBasedFrameDecoder` 与 `StringDecoder` 组合

`LineBasedFrameDecoder` 原理是检测字节数组中含有的“`\n`”或者“`\r\n`”换行符,并以此作为字节数据包的结束位置。`StringDecoder` 的作用是将接收到的字节数组转换成字符串,然后调用后面的处理器 `Handler`。需要注意的是,若我们的业务数据中存在换行符,需要做转义处理。

下面通过相应的代码示例来说明 `LineBasedFrameDecoder+StringDecoder` 组合的使用方法。

Netty 服务端 `LineBaseServer` 代码实现如下:

```
public class LineBaseServer {

    public static void main(String[] args) throws Exception {
        int port = 8080;
        new LineBaseServer().bind(port);
    }

    public void bind(int port) throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
              .channel(NioServerSocketChannel.class)
              .option(ChannelOption.SO_BACKLOG, 1024)
```

```
        .childHandler(new ChannelInitializer<NioSocketChannel>()
{
    @Override
    protected void initChannel(NioSocketChannel ch) throws
Exception {
        //设置 LineBasedFrameDecoder 处理器
        ch.pipeline().addLast(new
LineBasedFrameDecoder(1024));

        //设置 StringDecoder 处理器
        ch.pipeline().addLast(new StringDecoder());
        ch.pipeline().addLast(new
LineBaseServerHandler());
    }
});

    ChannelFuture f = b.bind(port).sync();
    f.channel().closeFuture().sync();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
}
```

服务端业务处理器 `LineBaseServerHandler` 实现如下:

```
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import java.util.concurrent.atomic.AtomicInteger;

public class LineBaseServerHandler extends SimpleChannelInboundHandler {
```



```

//计数器
private static final AtomicInteger counter = new AtomicInteger(0);

@Override
public void channelRead0(ChannelHandlerContext ctx, Object msg) throws
Exception {
    //接收客户端发送的字符串，并打印到控制台
    String content = (String) msg;
    System.out.println("received from client:" + content + " counter:"
+ counter.addAndGet(1));

    //将数据重新发送到客户端
    content += "\n";
    ByteBuf echo = Unpooled.copiedBuffer(content.getBytes());
    ctx.writeAndFlush(echo);
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    ctx.close();
}

@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
    ctx.flush();
}
}

```

Netty 客户端 LineBaseClient 实现代码如下:

```

import io.netty.bootstrap.Bootstrap;
import io.netty.buffer.ByteBuf;

```

```
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;

public class LineBaseClient {

    public static void main(String[] args) throws Exception {
        int port = 8080;
        new LineBaseClient().connect(port, "127.0.0.1");
    }

    public void connect(int port, String host) throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();
            b.group(group)
                .channel(NioSocketChannel.class)
                .option(ChannelOption.TCP_NODELAY, true)
                .handler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) throws
Exception {

                        //设置 LineBasedFrameDecoder 处理器
                        ch.pipeline().addLast(new
LineBasedFrameDecoder(1024));

                        //设置 StringDecoder 处理器
                        ch.pipeline().addLast(new StringDecoder());
                    }
                });
        } catch {
            group.shutdownGracefully();
        }
    }
}
```

```

        ch.pipeline().addLast(new
LineBaseClientHandler());
    }
});

ChannelFuture f = b.connect(host, port).sync();

//循环发送1000次
for (int i = 0; i < 1000; i++) {
    //构造客户端发送的数据 ByteBuf 对象
    String content = "你好,Netty!\n";
    byte[] req = content.getBytes();
    ByteBuf messageBuffer = Unpooled.buffer(req.length);
    messageBuffer.writeBytes(req);

    //向服务端发送数据
    ChannelFuture channelFuture =
f.channel().writeAndFlush(messageBuffer);
    channelFuture.syncUninterruptibly();
}
f.channel().closeFuture().sync();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    group.shutdownGracefully();
}
}
}

```

客户端业务处理器 **LineBaseClientHandler** 实现如下:

```

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;

```

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Logger;

public class LineBaseClientHandler extends SimpleChannelInboundHandler {

    private static final Logger logger =
        Logger.getLogger(LineBaseClientHandler.class.getName());
    //计数器
    private static final AtomicInteger counter = new AtomicInteger(0);

    @Override
    public void channelRead0(ChannelHandlerContext ctx, Object msg) throws
        Exception {
        //获取服务端返回的数据，并打印到控制台
        String content = (String) msg;
        System.out.println("received from server:" + content + " counter:"
+ counter.addAndGet(1));
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
        throws Exception {
        logger.warning("Unexpected exception from downstream : " +
            cause.getMessage());
        ctx.close();
    }
}
```

先运行 LineBaseServer 中的 main 方法，将 Netty 服务启动起来，再运行 LineBaseClient 中的 main 方法，循环发送 1000 条字符串给服务端，运行结果如下。

服务端控制台打印内容：

```
received from client:你好,Netty! counter:1
```

```
received from client:你好,Netty! counter:2
received from client:你好,Netty! counter:3
received from client:你好,Netty! counter:4
received from client:你好,Netty! counter:5
received from client:你好,Netty! counter:6
received from client:你好,Netty! counter:7
received from client:你好,Netty! counter:8
received from client:你好,Netty! counter:9
.....省略.....
.....省略.....
received from client:你好,Netty! counter:989
received from client:你好,Netty! counter:990
received from client:你好,Netty! counter:991
received from client:你好,Netty! counter:992
received from client:你好,Netty! counter:993
received from client:你好,Netty! counter:994
received from client:你好,Netty! counter:995
received from client:你好,Netty! counter:996
received from client:你好,Netty! counter:997
received from client:你好,Netty! counter:998
received from client:你好,Netty! counter:999
received from client:你好,Netty! counter:1000
```

客户端控制台打印内容:

```
received from server:你好,Netty! counter:1
received from server:你好,Netty! counter:2
received from server:你好,Netty! counter:3
received from server:你好,Netty! counter:4
received from server:你好,Netty! counter:5
received from server:你好,Netty! counter:6
received from server:你好,Netty! counter:7
received from server:你好,Netty! counter:8
received from server:你好,Netty! counter:9
```

```
.....省略.....  
.....省略.....  
received from server:你好,Netty! counter:992  
received from server:你好,Netty! counter:993  
received from server:你好,Netty! counter:994  
received from server:你好,Netty! counter:995  
received from server:你好,Netty! counter:996  
received from server:你好,Netty! counter:997  
received from server:你好,Netty! counter:998  
received from server:你好,Netty! counter:999  
received from server:你好,Netty! counter:1000
```

可以发现，无论是客户端还是服务端的控制台都完整地打印了 1000 条字符串，说明通过 `LineBasedFrameDecoder+StringDecoder` 组合已经解决了半包/粘包问题。

3. `FixedLengthFrameDecoder` 与 `StringDecoder` 组合

`FixedLengthFrameDecoder` 的原理是按照事先指定的固定长度对消息进行解码，通过这种方式解决粘包/半包问题。指定的固定长度为所传输消息的字节数组的长度。

下面通过相应的代码示例来说明 `FixedLengthFrameDecoder` 与 `StringDecoder` 组合的使用。

Netty 服务端 `FixedLengthServer` 实现：

```
public class FixedLengthServer {  
  
    public static void main(String[] args) throws Exception {  
        int port = 8080;  
        new FixedLengthServer().bind(port);  
    }  
  
    public void bind(int port) throws Exception {  
        EventLoopGroup bossGroup = new NioEventLoopGroup();  
        EventLoopGroup workerGroup = new NioEventLoopGroup();
```

```

try {
    ServerBootstrap b = new ServerBootstrap();
    b.group(bossGroup, workerGroup)
        .channel(NioServerSocketChannel.class)
        .option(ChannelOption.SO_BACKLOG, 1024)
        .childHandler(new ChannelInitializer<NioSocketChannel>()
        {
            @Override
            protected void initChannel(NioSocketChannel ch) throws
            Exception {
                //设置 FixedLengthFrameDecoder 处理器, 13 为所需要传输
                的字符串"你好, Netty!"的字节数组长度
                ch.pipeline().addLast(new
                FixedLengthFrameDecoder(13));
                //设置 StringDecoder 处理器
                ch.pipeline().addLast(new StringDecoder());
                ch.pipeline().addLast(new
                FixedLengthServerHandler());
            }
        });

    ChannelFuture f = b.bind(port).sync();
    f.channel().closeFuture().sync();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
}
}

```

服务端业务处理器 FixedLengthServerHandler 实现：

```
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import java.util.concurrent.atomic.AtomicInteger;

public class FixedLengthServerHandler extends SimpleChannelInboundHandler {
    //计数器
    private static final AtomicInteger counter = new AtomicInteger(0);

    @Override
    public void channelRead0(ChannelHandlerContext ctx, Object msg) throws
Exception {
        //接收客户端发送的字符串，并打印到控制台
        String content = (String) msg;
        System.out.println("received from client:" + content + " counter:"
+ counter.addAndGet(1));

        //将数据重新发送到客户端
        ByteBuf echo = Unpooled.copiedBuffer(content.getBytes());
        ctx.writeAndFlush(echo);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        ctx.close();
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws
```



```

Exception {
    ctx.flush();
}
}

```

Netty 客户端 FixedLengthClient 实现:

```

import io.netty.bootstrap.Bootstrap;
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.FixedLengthFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;

public class FixedLengthClient {

    public static void main(String[] args) throws Exception {
        int port = 8080;
        new FixedLengthClient().connect(port, "127.0.0.1");
    }

    public void connect(int port, String host) throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();
            b.group(group)
              .channel(NioSocketChannel.class)
              .option(ChannelOption.TCP_NODELAY, true)
              .handler(new ChannelInitializer<NioSocketChannel>() {

```

```
        @Override
        protected void initChannel(NioSocketChannel ch) throws
Exception {

            //设置 FixedLengthFrameDecoder 处理器, 13 为所需要传输
            的字符串"你好,Netty!"的字节数组长度
            ch.pipeline().addLast(new
FixedLengthFrameDecoder(13));

            //设置 StringDecoder 处理器
            ch.pipeline().addLast(new StringDecoder());
            ch.pipeline().addLast(new
FixedLengthClientHandler());
        }
    });

    //发起异步连接操作
    ChannelFuture f = b.connect(host, port).sync();

    //循环发送 1000 次
    for (int i = 0; i < 1000; i++) {
        //构造客户端发送的数据 ByteBuf 对象
        String content = "你好,Netty!";
        byte[] req = content.getBytes();
        ByteBuf messageBuffer = Unpooled.buffer(req.length);
        messageBuffer.writeBytes(req);

        //向服务端发送数据
        ChannelFuture channelFuture =
f.channel().writeAndFlush(messageBuffer);
        channelFuture.syncUninterruptibly();
    }
    f.channel().closeFuture().sync();
} catch (Exception e) {
    e.printStackTrace();
}
```

```

        } finally {
            group.shutdownGracefully();
        }
    }
}

```

客户端业务处理器 FixedLengthClientHandler:

```

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;

import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Logger;

public class FixedLengthClientHandler extends SimpleChannelInboundHandler
{
    private static final Logger logger =
        Logger.getLogger(FixedLengthClientHandler.class.getName());
    //计数器
    private static final AtomicInteger counter = new AtomicInteger(0);

    @Override
    public void channelRead0(ChannelHandlerContext ctx, Object msg) throws
Exception {
        //获取服务端返回的数据，并打印到控制台
        String content = (String) msg;
        System.out.println("received from server:" + content + " counter:"
+ counter.addAndGet(1));
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)

```

```
throws Exception {  
    logger.warning("Unexpected exception from downstream : " +  
cause.getMessage());  
    ctx.close();  
}  
}
```

先运行 `FixedLengthServer` 中的 `main` 方法，将 `Netty` 服务启动，再运行 `FixedLengthClient` 中的 `main` 方法，循环发送 1000 条字符串给服务端，运行结果如下。

服务端控制台打印内容：

```
received from client:你好,Netty! counter:1  
received from client:你好,Netty! counter:2  
received from client:你好,Netty! counter:3  
received from client:你好,Netty! counter:4  
received from client:你好,Netty! counter:5  
received from client:你好,Netty! counter:6  
received from client:你好,Netty! counter:7  
received from client:你好,Netty! counter:8  
received from client:你好,Netty! counter:9  
.....省略.....  
.....省略.....  
received from client:你好,Netty! counter:989  
received from client:你好,Netty! counter:990  
received from client:你好,Netty! counter:991  
received from client:你好,Netty! counter:992  
received from client:你好,Netty! counter:993  
received from client:你好,Netty! counter:994  
received from client:你好,Netty! counter:995  
received from client:你好,Netty! counter:996  
received from client:你好,Netty! counter:997  
received from client:你好,Netty! counter:998  
received from client:你好,Netty! counter:999
```

```
received from client:你好,Netty! counter:1000
```

客户端控制台打印内容:

```
received from server:你好,Netty! counter:1
received from server:你好,Netty! counter:2
received from server:你好,Netty! counter:3
received from server:你好,Netty! counter:4
received from server:你好,Netty! counter:5
received from server:你好,Netty! counter:6
received from server:你好,Netty! counter:7
received from server:你好,Netty! counter:8
received from server:你好,Netty! counter:9
.....省略.....
.....省略.....
received from server:你好,Netty! counter:992
received from server:你好,Netty! counter:993
received from server:你好,Netty! counter:994
received from server:你好,Netty! counter:995
received from server:你好,Netty! counter:996
received from server:你好,Netty! counter:997
received from server:你好,Netty! counter:998
received from server:你好,Netty! counter:999
received from server:你好,Netty! counter:1000
```

可以发现,无论是客户端还是服务端的控制台都完整地打印了 1000 条字符串,说明通过 FixedLengthFrameDecoder+StringDecoder 组合已经解决了半包/粘包问题。

6.2.2.2 Netty 内置的完整的编码解码方案

Netty 也内置了部分常用的编解码方案用来解决粘包/半包问题,常用的主要有以下几种方案。

- ◎ 通过解码组合 ProtobufDecoder/ProtobufVarint32FrameDecoderProtoBuf 及编码组

合 `ProtobufEncoder/ProtobufVarint32LengthFieldPrepender` 提供对 `protobuf` 的编解码支持。

- ◎ 通过 `ObjectDecoder` 与 `ObjectEncoder` 提供对 Java 内置序列化编解码支持。
- ◎ 通过 `MarshallDecoder` 与 `MarshallEncoder` 提供了 `Marshall` 序列化编解码支持，还有其他的，比如 `JSON`、`XML`、`Base64`、`HTTP`、`Memcache` 等编解码方案的支持。

1. protobuf 编解码内置方案

Netty 为 `protobuf` 序列化方式提供了完整的编解码支持。

解码工具类组合：

```
io.netty.handler.codec.protobuf.ProtobufDecoder
io.netty.handler.codec.protobuf.ProtobufVarint32FrameDecoder
```

编码工具类组合：

```
io.netty.handler.codec.protobuf.ProtobufEncoder
io.netty.handler.codec.protobuf.ProtobufVarint32LengthFieldPrepender
```

通过以上的编解码工具类能够有效地解决传输过程中出现的粘包/半包问题，解决原理如下。

- ◎ 编码过程：通过 `ProtobufVarint32LengthFieldPrepender` 将整个消息体的长度作为消息头附加在消息体，然后使用 `ProtobufDecoder` 进行编码，编码完成后将编码结果字节数组通过 Netty 进行传输。
- ◎ 解码过程：接受消息的时候，`ProtobufVarint32FrameDecoder` 先接受消息头，获取消息体的字节数组长度，直到获取到等于消息字节数组长度的字节数，才使用 `ProtobufDecoder` 进行解码操作。

下面通过具体的代码示例说明 Netty 使用 `protobuf` 序列化方式进行通信。

proto 描述文件源码:

```

syntax="proto2";

package netty.protobuf;
option java_package="netty.protobuf";
option java_outer_classname="UserInfo";

message User{
    required string name =1;
    required int64 userId =2;
    optional string email=3;
    optional string mobile=4;
    optional string remark=5;
}

```

编译之后, 生成类文件 netty.protobuf.UserInfo。

下面是 Netty 通信相关的代码。

服务端代码 ProtobufServer:

```

package netty.protobuf;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.protobuf.ProtobufDecoder;
import io.netty.handler.codec.protobuf.ProtobufVarint32FrameDecoder;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;

```

```
public class ProtobufServer {

    public void bind(int port) throws Exception {
        //配置服务端的NIO线程组
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                .channel(NioServerSocketChannel.class)
                .option(ChannelOption.SO_BACKLOG, 100)
                .handler(new LoggingHandler(LogLevel.INFO))
                .childHandler(new ChannelInitializer<SocketChannel>() {

                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        // 配置 Protobuf 解码工具 ProtobufVarint32FrameDecoder 与
ProtobufDecoder

                        ch.pipeline().addLast(new
ProtobufVarint32FrameDecoder());

                        ch.pipeline().addLast(new
ProtobufDecoder(UserInfo.User.getDefaultInstance()));

                        ch.pipeline().addLast(new ProtoServerHandler());
                    }
                });

            //绑定端口，同步等待成功
            ChannelFuture f = b.bind(port).sync();

            //等待服务端监听端口关闭
            f.channel().closeFuture().sync();
        }
    }
}
```



```

    } finally {
        //优雅退出, 释放线程池资源
        bossGroup.shutdownGracefully();
        workerGroup.shutdownGracefully();
    }
}

public static void main(String[] args) throws Exception {
    int port = 8080;
    new ProtobufServer().bind(port);
}
}

```

服务端接收客户端发送的消息代码 ProtoServerHandler:

```

package netty.protobuf;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

public class ProtoServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        //接收客户端发送过来的消息, 其中经过前面解码工具 ProtobufVarint32FrameDecoder 与
        //ProtobufDecoder 的处理, 将字节码消息自动转换成了 UserInfo.User 对象
        UserInfo.User req = (UserInfo.User) msg;
        System.out.println("received from client:" + req.toString());
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
{

```

```
        cause.printStackTrace();
        ctx.close();
    }
}
```

客户端代码 ProtobufClient:

```
package netty.protobuf;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.protobuf.ProtobufEncoder;
import
io.netty.handler.codec.protobuf.ProtobufVarint32LengthFieldPrepender;

public class ProtobufClient {

    public void connect(int port, String host) throws Exception {
        //配置客户端 NIO 线程组
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();
            b.group(group)
              .channel(NioSocketChannel.class)
              .option(ChannelOption.TCP_NODELAY, true)
              .handler(new ChannelInitializer<SocketChannel>() {
```

```

        @Override
        public void initChannel(SocketChannel ch) throws
Exception {
            //配置Protobuf 编码工具 ProtobufVarint32LengthFieldPrepender 与 ProtobufEncoder
            ch.pipeline().addLast(new
ProtobufVarint32LengthFieldPrepender());
            ch.pipeline().addLast(new ProtobufEncoder());
            ch.pipeline().addLast(new
ProtobufClientHandler());
        }
    });

    ChannelFuture f = b.connect(host, port).sync();
    f.channel().closeFuture().sync();
} finally {
    group.shutdownGracefully();
}
}

public static void main(String[] args) throws Exception {
    int port = 8080;
    new ProtobufClient().connect(port, "127.0.0.1");
}
}

```

客户端向服务端发送消息的代码 ProtobufClientHandler:

```

package netty.protobuf;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

public class ProtobufClientHandler extends ChannelInboundHandlerAdapter {

```

```
@Override
public void channelActive(ChannelHandlerContext ctx) {
    //channel 建立之后，向服务端发送消息，需要注意的是这里写入的消息是完整的
    UserInfo.User 对象
    //因为后续会被工具 ProtobufVarint32LengthFieldPrepender 与 ProtobufEncoder
    进行编码处理

    UserInfo.User user = UserInfo.User.newBuilder()
        .setName("kongxuan")
        .setUserId(10000)
        .setEmail("liyebing@163.com")
        .setMobile("153****0976")
        .setRemark("remark info").build();

    ctx.writeAndFlush(user);
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
{
    cause.printStackTrace();
    ctx.close();
}}
```

代码编写完毕之后。先运行 `ProtobufServer` 的 `main` 方法，将 `Netty` 服务端启动起来，然后运行 `ProtobufClient` 的 `main` 方法。在服务端的控制台会打印出如下信息：

```
received from client:name: "kongxuan"
userId: 10000
email: "liyebing@163.com"
mobile: "153****0976"
remark: "remark info"
```

2. Java 默认序列化方式编解码内置方案

Java 默认的序列化协议虽然性能比 protobuf 序列化协议要差，但其与 Java 语言天然的亲和性有其独有的优势，在不苛求性能的场景，Java 序列化也是不错的选择。

Netty 为 Java 默认的序列化方式提供了完整的编解码支持。

解码工具：

```
io.netty.handler.codec.serialization.ObjectDecoder
```

编码工具：

```
io.netty.handler.codec.serialization.ObjectEncoder
```

下面给出具体的代码示例。

服务端代码 `JavaSerializeServer`：

```
public class JavaSerializeServer {

    public void bind(int port) throws Exception {
        //配置服务端的NIO线程组
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
              .channel(NioServerSocketChannel.class)
              .option(ChannelOption.SO_BACKLOG, 100)
              .handler(new LoggingHandler(LogLevel.INFO))
              .childHandler(new ChannelInitializer<SocketChannel>() {
                  @Override
                  protected void initChannel(SocketChannel ch) throws
Exception {
```

```
        //配置 Java 默认序列化解码工具 ObjectDecoder
        ch.pipeline().addLast(new ObjectDecoder(100 *
1024,
        ClassResolvers.weakCaching
ConcurrentResolver(this.getClass().getClassLoader())));
        ch.pipeline().addLast(new
JavaSerializeServerHandler());
    }
    });

    //绑定端口，同步等待成功
    ChannelFuture f = b.bind(port).sync();
    //等待服务端监听端口关闭
    f.channel().closeFuture().sync();
} finally {
    //优雅退出，释放线程池资源
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
}

public static void main(String[] args) throws Exception {
    int port = 8080;
    new JavaSerializeServer().bind(port);
}
}
```

服务端处理业务逻辑 handler `JavaSerializeServerHandler`:

```
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

public class JavaSerializeServerHandler extends ChannelInboundHandlerAdapter
{
```

```

@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    //接收客户端发送过来的消息，其中经过前面解码工具 ObjectDecoder 的处理，将字节
    码消息自动转换成了 UserInfo 对象
    UserInfo req = (UserInfo) msg;
    System.out.println("received from client:" + req.toString());
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
{
    cause.printStackTrace();
    ctx.close();
}
}

```

客户端代码 JavaSerializeClient:

```

package netty.javaserialize;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.serialization.ObjectEncoder;

public class JavaSerializeClient {

```

```
public void connect(int port, String host) throws Exception {
    //配置客户端 NIO 线程组
    EventLoopGroup group = new NioEventLoopGroup();
    try {
        Bootstrap b = new Bootstrap();
        b.group(group)
            .channel(NioSocketChannel.class)
            .option(ChannelOption.TCP_NODELAY, true)
            .handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws
Exception {
                    //配置 Java 默认序列化解码工具 ObjectEncoder
                    ch.pipeline().addLast(new ObjectEncoder());
                    ch.pipeline().addLast(new
JavaSerializeClientHandler());
                }
            });

        ChannelFuture f = b.connect(host, port).sync();
        f.channel().closeFuture().sync();
    } finally {
        group.shutdownGracefully();
    }
}

public static void main(String[] args) throws Exception {
    int port = 8080;
    new JavaSerializeClient().connect(port, "127.0.0.1");
}
}
```

客户端业务逻辑 handler `JavaSerializeClientHandler`:

```
package netty.javaserialize;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

public class JavaSerializeClientHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        //channel 建立之后, 向服务端发送消息, 需要注意的是这里写入的消息是完整的
        //UserInfo 对象
        //因为后续会被 ObjectEncoder 进行编码处理
        UserInfo user = UserInfo.newBuilder()
            .name("liyebing")
            .userId(10000)
            .email("liyebing@163.com")
            .mobile("153****0976")
            .remark("remark info").build();

        ctx.writeAndFlush(user);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
    {
        cause.printStackTrace();
        ctx.close();
    }
}
```

先运行 `JavaSerializeServer` 类的 `main` 方法，将 `Netty` 服务端启动起来，然后运行 `JavaSerializeClient` 类的 `main` 方法，发现在服务端的控制台会打印如下信息：

```
received      from      client:UserInfo{name='liyebing',      userId=10000,
email='liyebing@163.com', mobile='153****0976', remark='remark info'
```

3. JBoss Marshalling 序列化编解码内置方案

`Netty` 也为 `JBoss Marshalling` 序列化提供了内置的编解码解决方案，能够有效解决粘包与半包问题。

解码工具：

```
io.netty.handler.codec.marshalling.MarshallingDecoder
```

编码工具：

```
io.netty.handler.codec.marshalling.MarshallingEncoder
```

下面通过具体的代码示例来说明如何使用 `Netty` 提供的 `Marshalling` 编解码工具。

服务端 `MarshallingSerializeServer`：

```
public class MarshallingSerializeServer {

    public void bind(int port) throws Exception {
        //配置服务端的NIO线程组
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
              .channel(NioServerSocketChannel.class)
              .option(ChannelOption.SO_BACKLOG, 100)
              .handler(new LoggingHandler(LogLevel.INFO))
              .childHandler(new ChannelInitializer<SocketChannel>() {
```

```

@Override
protected void initChannel(SocketChannel ch) throws
Exception {

    //配置 Marshalling 序列化解码工具
    ch.pipeline().addLast(MarshallingCodeCFactory.
buildMarshallingDecoder());

    ch.pipeline().addLast(new
MarshallingSerializeServerHandler());
}

});

//绑定端口, 同步等待成功
ChannelFuture f = b.bind(port).sync();
//等待服务端监听端口关闭
f.channel().closeFuture().sync();
} finally {
    //优雅退出, 释放线程池资源
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
}

public static void main(String[] args) throws Exception {
    int port = 8080;
    new MarshallingSerializeServer().bind(port);
}
}

```

服务端业务逻辑 Handler MarshallingSerializeServerHandler:

```

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

```

```

public class MarshallingSerializeServerHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        //接收客户端发送过来的消息，其中经过前面解码工具的处理，将字节码消息自动转换成
了UserInfo 对象
        UserInfo req = (UserInfo) msg;
        System.out.println("received from client:" + req.toString());
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
{
        cause.printStackTrace();
        ctx.close();
    }
}

```

客户端 MarshallingSerializeClient:

```

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;

public class MarshallingSerializeClient {

    public void connect(int port, String host) throws Exception {

```

```

//配置客户端 NIO 线程组
EventLoopGroup group = new NioEventLoopGroup();
try {
    Bootstrap b = new Bootstrap();
    b.group(group)
        .channel(NioSocketChannel.class)
        .option(ChannelOption.TCP_NODELAY, true)
        .handler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws
Exception {
                //配置 Marshalling 序列化编码工具

                ch.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());
                ch.pipeline().addLast(new
MarshallingSerializeClientHandler());
            }
        });

    ChannelFuture f = b.connect(host, port).sync();
    f.channel().closeFuture().sync();
} finally {
    group.shutdownGracefully();
}

public static void main(String[] args) throws Exception {
    int port = 8080;
    new MarshallingSerializeClient().connect(port, "127.0.0.1");
}
}

```

客户端业务逻辑 handler MarshallingSerializeClientHandler:

```
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

public class MarshallingSerializeClientHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        //channel 建立之后，向服务端发送消息，需要注意的是这里写入的消息是完整的
        //UserInfo 对象
        UserInfo user = UserInfo.newBuilder()
            .name("liyebing")
            .userId(10000)
            .email("liyebing@163.com")
            .mobile("153****0976")
            .remark("remark info").build();

        ctx.writeAndFlush(user);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
    {
        cause.printStackTrace();
        ctx.close();
    }
}
```

序列化编解码工具类：

```
import io.netty.handler.codec.marshalling.*;
import org.jboss.marshalling.MarshallerFactory;
```

```

import org.jboss.marshalling.Marshalling;
import org.jboss.marshalling.MarshallingConfiguration;

public class MarshallingCodeCFactory {

    //首先通过 Marshalling 序列化工厂类，参数 serial 标识创建的是 Java 序列化工厂对象
    private final static MarshallerFactory marshallerFactory =
Marshalling.getProvidedMarshallerFactory("serial");

    //创建了 MarshallingConfiguration 对象，配置了版本号为 5
    private static final MarshallingConfiguration configuration = new
MarshallingConfiguration();

    static {
        configuration.setVersion(5);
    }

    /**
     * 创建 Jboss Marshalling 解码器 MarshallingDecoder
     *
     * @return MarshallingDecoder
     */
    public static MarshallingDecoder buildMarshallingDecoder() {
        //根据 marshallerFactory 和 configuration 创建 provider
        UnmarshallerProvider provider = new
DefaultUnmarshallerProvider(marshallerFactory, configuration);
        //构建 Netty 的 MarshallingDecoder 对象，两个参数分别为 provider 和单个消息
序列化后的最大长度
        return new MarshallingDecoder(provider, 1024 * 100);
    }

    /**
     * 创建 Jboss Marshalling 编码器 MarshallingEncoder
     *

```

```

        * @return MarshallingEncoder
        */
        public static MarshallingEncoder buildMarshallingEncoder() {
            MarshallerProvider provider = new
DefaultMarshallerProvider(marshallerFactory, configuration);
            //构建 Netty 的 MarshallingEncoder 对象, MarshallingEncoder 用于实现序列化接口的 POJO 对象序列化为二进制数组
            return new MarshallingEncoder(provider);
        }
    }
}

```

其中, 需要依赖 Marshalling 相关的 jar 包, Maven 配置如下:

```

<dependency>
<groupId>org.jboss.marshalling</groupId>
<artifactId>jboss-marshalling-serial</artifactId>
<version>2.0.0.Beta2</version>
</dependency>

```

先运行 MarshallingSerializeServer 类中的 main 方法, 将 Netty 服务端启动起来。然后运行 MarshallingSerializeClient 中的 main 方法, 发起客户端调用。运行结果如下:

```

received from client:UserInfo{name='liyebing', userId=10000, email='liyebing@163.com', mobile='153****0976', remark='remark info'}

```

6.2.2.3 Netty 自定义编解码器开发

除了使用 Netty 内置的编解码方案, 我们也可以自定义编解码器来解决粘包半包问题。下面介绍如何使用 Netty 中的 MessageToByteEncoder 与 ByteToMessageDecoder 来自定义编解码器。

其原理是使用 int 数据类型来记录整个消息的字节数组长度, 将该 int 数据作为消息的消息头一起传输, 在服务端接收消息数据的时候, 先接收 4 个字节的 int 数据类型数据, 这个数据即为整个消息字节数组的长度, 再接收剩余字节, 直到接收的字节数组长度等于

最先接收的 `int` 数据类型数据大小。

消息格式如下：

length Message Data

1. 使用 `LengthFieldPrepender` 与 `LengthFieldBasedFrameDecoder`

在编码器之前增加 `LengthFieldPrepender`，将在消息体字节数组之前新增几个字节，来表示消息的长度。其原理如下：

编码之前	编码之后
+-----+	+-----+
消息内容 ----->	长度 消息内容
"HELLO, WORLD"	0x000C "HELLO, WORLD"
+-----+	+-----+

在解码器之前使用 `LengthFieldBasedFrameDecoder` 来处理半包与粘包消息，它可以根据消息头部标识的消息长度来分包，使得后面的解码器接收到的消息是整包消息，不会出现半包或者粘包的情况。其原理如下：

解码之前	解码之后
+-----+	+-----+
长度 消息内容 ----->	消息内容
0x000C "HELLO, WORLD"	"HELLO, WORLD"
+-----+	+-----+

下面我们将通过代码示例来演示 `LengthFieldPrepender` 与 `LengthFieldBasedFrameDecoder` 的使用。

服务端实现 `CustomServer`：

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
```

```
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.LengthFieldBasedFrameDecoder;
import io.netty.handler.codec.LengthFieldPrepender;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;

public class CustomServer {

    public void bind(int port) throws Exception {
        //配置服务端的 NIO 线程组
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                .channel(NioServerSocketChannel.class)
                .option(ChannelOption.SO_BACKLOG, 100)
                .handler(new LoggingHandler(LogLevel.INFO))
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        //解决粘包/半包，根据消息长度自动拆包
                        ch.pipeline().addLast(new
LengthFieldBasedFrameDecoder(65535, 0, 2, 0, 2));
                        //配置自定义序列化解码工具
                        ch.pipeline().addLast(new CustomV1Decoder());
                        //解决粘包/半包问题附加消息长度在消息头部
                        ch.pipeline().addLast(new
```

```

LengthFieldPrepender(2));

        //配置自定义序列化编码工具
        ch.pipeline().addLast(new CustomVlEncoder());
        ch.pipeline().addLast(new CustomServerHandler());
    }

});

//绑定端口, 同步等待成功
ChannelFuture f = b.bind(port).sync();
//等待服务端监听端口关闭
f.channel().closeFuture().sync();
} finally {
    //优雅退出, 释放线程池资源
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
}

public static void main(String[] args) throws Exception {
    int port = 8080;
    new CustomServer().bind(port);
}
}

```

服务端业务逻辑的实现 CustomServerHandler:

```

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import java.util.concurrent.atomic.AtomicInteger;

public class CustomServerHandler extends ChannelInboundHandlerAdapter {

    private static AtomicInteger counter = new AtomicInteger(0);

```

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    //接收客户端发送过来的消息，其中经过前面解码工具的处理，将字节码消息自动转换成
    了 UserInfo 对象
    UserInfo req = (UserInfo) msg;
    System.out.println("received from client:" + req.toString() + "
counter :" + counter.incrementAndGet());
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
{
    cause.printStackTrace();
    ctx.close();
}
}
```

客户端实现 CustomClient:

```
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.LengthFieldBasedFrameDecoder;
import io.netty.handler.codec.LengthFieldPrepender;

public class CustomClient {
```

```

public void connect(int port, String host) throws Exception {
    //配置客户端 NIO 线程组
    EventLoopGroup group = new NioEventLoopGroup();
    try {
        Bootstrap b = new Bootstrap();
        b.group(group)
            .channel(NioSocketChannel.class)
            .option(ChannelOption.TCP_NODELAY, true)
            .handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws
Exception {
                    //解决粘包/半包, 根据消息长度自动拆包
                    ch.pipeline().addLast(new
LengthFieldBasedFrameDecoder(65535, 0, 2, 0, 2));
                    //配置自定义序列化解码工具
                    ch.pipeline().addLast(new CustomV1Decoder());
                    //解决粘包/半包问题附加消息长度在消息头部
                    ch.pipeline().addLast(new
LengthFieldPrepender(2));
                    //配置自定义序列化编码工具
                    ch.pipeline().addLast(new CustomV1Encoder());
                    ch.pipeline().addLast(new CustomClientHandler());
                }
            });

        ChannelFuture f = b.connect(host, port).sync();
        f.channel().closeFuture().sync();
    } finally {
        group.shutdownGracefully();
    }
}

```

```
public static void main(String[] args) throws Exception {  
    int port = 8080;  
    new CustomClient().connect(port, "127.0.0.1");  
}  
}
```

客户端业务逻辑 handler 实现 CustomClientHandler:

```
import io.netty.channel.ChannelHandlerContext;  
import io.netty.channel.ChannelInboundHandlerAdapter;  
  
public class CustomClientHandler extends ChannelInboundHandlerAdapter {  
  
    @Override  
    public void channelActive(ChannelHandlerContext ctx) {  
        for (int i = 0; i < 1000; i++) {  
            //channel 建立之后，向服务端发送消息，需要注意的是这里写入的消息是完整的  
            UserInfo 对象  
            UserInfo user = UserInfo.newBuilder()  
                .name("liyebing")  
                .userId(10000)  
                .email("liyebing@163.com")  
                .mobile("153****0976")  
                .remark("remark info").build();  
  
            ctx.writeAndFlush(user);  
        }  
    }  
  
    @Override  
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)  
    {  
        cause.printStackTrace();  
    }  
}
```

```

        ctx.close();
    }
}

```

编码器实现 CustomV1Encoder:

```

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.MessageToByteEncoder;

public class CustomV1Encoder extends MessageToByteEncoder {
    @Override
    public void encode(ChannelHandlerContext ctx, Object in, ByteBuf out)
throws Exception {
        //使用 hessian 序列化对象
        byte[] data = HessianSerializer.serialize(in);
        out.writeBytes(data);
    }
}

```

解码器实现 CustomV1Decoder:

```

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ByteToMessageDecoder;

import java.util.List;

public class CustomV1Decoder extends ByteToMessageDecoder {

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {

```

```
//读取字节数组
int dataLength = in.readableBytes();
if (dataLength <= 0) {
    return;
}
byte[] data = new byte[dataLength];
in.readBytes(data);

//将字节数组使用 Hessian 反序列化为对象
Object obj = HessianSerializer.deserialize(data);
out.add(obj);
}
}
```

其中编解码器序列化采用了 Hessian 序列化。HessianSerializer 实现如下：

```
import com.caucho.hessian.io.HessianInput;
import com.caucho.hessian.io.HessianOutput;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

public class HessianSerializer {

    private HessianSerializer() {
    }

    public static byte[] serialize(Object obj) {
        if (obj == null)
            throw new NullPointerException();

        try {
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            HessianOutput ho = new HessianOutput(os);
            ho.writeObject(obj);
        }
    }
}
```



```

        return os.toByteArray();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

@SuppressWarnings("unchecked")
public static <T> T deserialize(byte[] data) {
    if (data == null)
        throw new NullPointerException();

    try {
        ByteArrayInputStream is = new ByteArrayInputStream(data);
        HessianInput hi = new HessianInput(is);
        return (T) hi.readObject();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

先运行 CustomServer 的 main 方法，将 Netty 服务端启动起来。然后运行 CustomClient 客户端 main 方法发起调用。CustomClientHandler 中当 channel 建立连接之后，调用 channelActive 方法向服务端循环写入 1000 个 UserInfo 对象的值。服务端 CustomServerHandler 的 channelRead 方法会将接收到的对象打印到控制台。

运行结果如下：

```

received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :1
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :2
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :3

```

```
received from client:UserInfo{name='liyebing', userId=10000, email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :4
received from client:UserInfo{name='liyebing', userId=10000, email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :5
.....省略.....
.....省略.....
received from client:UserInfo{name='liyebing', userId=10000, email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :998
received from client:UserInfo{name='liyebing', userId=10000, email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :999
received from client:UserInfo{name='liyebing', userId=10000, email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :1000
```

发现完整地打印了 1000 个对象的值，由此可以证明已经解决了半包或者粘包的问题。

2. 另一种构建自定义编解码器的方式

我们也可以不用 Netty 为我们提供的 LengthFieldPrepender 与 LengthFieldBasedFrame Decoder 工具类，完全由自己实现类似的机制。

其实现代码大部分类似，下面给出不同的关键代码实现。

编码器 CustomV2Encoder:

```
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.MessageToByteEncoder;

public class CustomV2Encoder extends MessageToByteEncoder {

    @Override
    public void encode(ChannelHandlerContext ctx, Object in, ByteBuf out)
        throws Exception {
        //使用 hessian 序列化对象
        byte[] data = HessianSerializer.serialize(in);
```

```

        //先写入消息的长度作为消息头
        out.writeInt(data.length);
        //最后写入消息体字节数组
        out.writeBytes(data);
    }
}

```

解码器 CustomV2Decoder:

```

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ByteToMessageDecoder;

import java.util.List;

public class CustomV2Decoder extends ByteToMessageDecoder {

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        //读消息头, 整个消息的长度字段
        if (in.readableBytes() < 4) {
            return;
        }
        in.markReaderIndex();
        int dataLength = in.readInt();
        if (dataLength < 0) {
            ctx.close();
        }
        //读取字节数组, 直到读取的字节数组长度等于 dataLength
        if (in.readableBytes() < dataLength) {
            in.resetReaderIndex();
            return;
        }
    }
}

```

```

byte[] data = new byte[dataLength];
in.readBytes(data);

//将字节数组使用 Hessian 反序列化为对象
Object obj = HessianSerializer.deserialize(data);
out.add(obj);
}
}

```

Netty 服务端 CustomServer 关键代码实现如下：

```

.....省略.....
ServerBootstrap b = new ServerBootstrap();
    b.group(bossGroup, workerGroup)
      .channel(NioServerSocketChannel.class)
      .option(ChannelOption.SO_BACKLOG, 100)
      .handler(new LoggingHandler(LogLevel.INFO))
      .childHandler(new ChannelInitializer<SocketChannel>() {
          @Override
          protected void initChannel(SocketChannel ch) throws
Exception {

                //配置自定义序列化解码工具
                ch.pipeline().addLast(new CustomV2Decoder());
                //配置自定义序列化编码工具
                ch.pipeline().addLast(new CustomV2Encoder());

                ch.pipeline().addLast(new CustomServerHandler());
            }
        });

    //绑定端口，同步等待成功
    ChannelFuture f = b.bind(port).sync();
    //等待服务端监听端口关闭
    f.channel().closeFuture().sync();

```

.....省略.....

Netty 客户端 CustomClient 关键代码实现如下:

```
.....省略.....
Bootstrap b = new Bootstrap();
    b.group(group)
      .channel(NioSocketChannel.class)
      .option(ChannelOption.TCP_NODELAY, true)
      .handler(new ChannelInitializer<SocketChannel>() {
          @Override
          public void initChannel(SocketChannel ch) throws
Exception {
                //配置自定义序列化解码工具
                ch.pipeline().addLast(new CustomV2Decoder());
                //配置自定义序列化编码工具
                ch.pipeline().addLast(new CustomV2Encoder());
                ch.pipeline().addLast(new CustomClientHandler());
            }
        });

    ChannelFuture f = b.connect(host, port).sync();
    f.channel().closeFuture().sync();
.....省略.....
```

先运行 CustomServer 的 main 方法, 将 Netty 服务端启动起来。然后运行 CustomClient 客户端 main 方法发起调用。CustomClientHandler 中当 channel 建立连接之后, 调用 channelActive 方法向服务端循环写入 1000 个 UserInfo 对象的值。在服务端 CustomServerHandler 的 channelRead 方法会将接收到的对象打印到控制台。

运行结果如下:

```
received from client:UserInfo{name='liyebing', userId=10000, email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :1
```

```
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :2
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :3
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :4
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :5
.....省略.....
.....省略.....
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :998
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :999
received from client:UserInfo{name='liyebing',   userId=10000,   email=
'liyebing@163.com', mobile='153****0976', remark='remark info'} counter :1000
```

发现完整地打印了 1000 个对象的值，由此可以证明通过另外一种方式也解决了半包或者粘包的问题。

6.3 使用 Netty 构建服务框架底层通信

前面花了大量的篇幅介绍了 Java I/O 模型及 I/O 类库相关的知识。同时也针对开源 NIO 通信框架 Netty 的使用做了介绍。本节我们将介绍在构建分布式服务框架过程中 Netty 的使用。

6.3.1 构建分布式服务框架 Netty 服务端

Netty 服务端的作用是提供启动 Netty 服务的方法，相对应的编解码处理器及服务端业

务逻辑处理器。下面通过具体的代码来说明相关的实现细节。

```

public class NettyServer {

    private static NettyServer nettyServer = new NettyServer();

    //服务端 boss 线程组
    private EventLoopGroup bossGroup;
    //服务端 worker 线程组
    private EventLoopGroup workerGroup;
    //序列化类型配置信息
    private SerializeType serializeType = PropertyConfigHelper.
getSerializeType();

    /**
     * 启动 Netty 服务
     *
     * @param port
     */
    public void start(final int port) {
        synchronized (NettyServer.class) {
            if (bossGroup != null || workerGroup != null) {
                return;
            }

            bossGroup = new NioEventLoopGroup();
            workerGroup = new NioEventLoopGroup();
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap

                .group(bossGroup, workerGroup)
                .channel(NioServerSocketChannel.class)
                .option(ChannelOption.SO_BACKLOG, 1024)
                .childOption(ChannelOption.SO_KEEPALIVE, true)

```

```
.childOption(ChannelOption.TCP_NODELAY, true)
.handler(new LoggingHandler(LogLevel.INFO))
.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws
Exception {

        //注册解码器 NettyDecoderHandler
        ch.pipeline().addLast(new
NettyDecoderHandler(AresRequest.class, serializeType));
        //注册编码器 NettyEncoderHandler
        ch.pipeline().addLast(new
NettyEncoderHandler(serializeType));
        //注册服务端业务逻辑处理器 NettyServerInvokeHandler
        ch.pipeline().addLast(new
NettyServerInvokeHandler());
    }
});

try {
    serverBootstrap.bind(port).sync().channel();
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}

}

private NettyServer() {
}

public static NettyServer singleton() {
    return nettyServer;
}
```



```

    }
}

```

其中 NettyDecoderHandler 为解码器，负责将字节数组解码为 Java 对象。具体实现如下：

```

import ares.remoting.framework.serialization.common.SerializeType;
import ares.remoting.framework.serialization.engine.SerializerEngine;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ByteToMessageDecoder;
import java.util.List;

public class NettyDecoderHandler extends ByteToMessageDecoder {

    //解码对象 class
    private Class<?> genericClass;
    //解码对象编码所使用序列化类型
    private SerializeType serializeType;

    public NettyDecoderHandler(Class<?> genericClass, SerializeType
serializeType) {
        this.genericClass = genericClass;
        this.serializeType = serializeType;
    }

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        //获取消息头所标识的消息体字节数组长度
        if (in.readableBytes() < 4) {
            return;
        }
        in.markReaderIndex();
    }
}

```

```
int dataLength = in.readInt();
if (dataLength < 0) {
    ctx.close();
}
//若当前可以获取到的字节数小于实际长度，则直接返回，直到当前可以获取到的字节数
等于实际长度
if (in.readableBytes() < dataLength) {
    in.resetReaderIndex();
    return;
}
//读取完整的消息体字节数组
byte[] data = new byte[dataLength];
in.readBytes(data);

//将字节数组反序列化为 Java 对象（SerializerEngine 参考序列化与反序列化章节）
Object obj = SerializerEngine.deserialize(data, genericClass,
serializeType.getSerializeType());
out.add(obj);
}
}
```

其中 `NettyEncoderHandler` 为编码器，负责将 Java 对象序列化为字节数组。具体实现如下：

```
import ares.remoting.framework.serialization.common.SerializeType;
import ares.remoting.framework.serialization.engine.SerializerEngine;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.MessageToByteEncoder;

public class NettyEncoderHandler extends MessageToByteEncoder {
    //序列化类型
    private SerializeType serializeType;
```

```

public NettyEncoderHandler(SerializeType serializeType) {
    this.serializeType = serializeType;
}

@Override
public void encode(ChannelHandlerContext ctx, Object in, ByteBuf out)
throws Exception {
    //将对象序列化为字节数组
    byte[] data = SerializerEngine.serialize(in, serializeType.
getSerializeType());
    //将字节数组（消息体）的长度作为消息头写入，解决半包/粘包问题
    out.writeInt(data.length);
    //写入序列化后得到的字节数组
    out.writeBytes(data);
}
}

```

Netty 服务端接收客户端发起的请求字节数组，然后通过解码 NettyDecoderHandler 将字节数组解码为对应的 Java 请求对象。下面需要做的事情就是根据解码得到的 Java 请求对象确定服务提供者接口及方法，然后通过使用 Java 反射来发起调用。

下面通过具体的代码来解读实现细节。

```

/**
 * 处理服务端的逻辑
 *
 */
@ChannelHandler.Sharable
public class NettyServerInvokeHandler extends SimpleChannelInboundHandler
<AresRequest> {

    private static final Logger logger = LoggerFactory.getLogger
(NettyServerInvokeHandler.class);

```

```
//服务端限流
private static final Map<String, Semaphore> serviceKeySemaphoreMap =
Maps.newConcurrentMap();

@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
    ctx.flush();
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    cause.printStackTrace();
    //发生异常，关闭链路
    ctx.close();
}

@Override
protected void channelRead0(ChannelHandlerContext ctx, AresRequest
request) throws Exception {

    if (ctx.channel().isWritable()) {
        //从服务调用对象里获取服务提供者信息
        ProviderService metaDataModel = request.getProviderService();
        long consumeTimeOut = request.getInvokeTimeout();
        final String methodName = request.getInvokedMethodName();

        //根据方法名称定位到具体某一个服务提供者
        String serviceKey = metaDataModel.getServiceItf().getName();
        //获取限流工具类
        int workerThread = metaDataModel.getWorkerThreads();
        Semaphore semaphore = serviceKeySemaphoreMap.get(serviceKey);
```

```

//初始化流控基础设施 semaphore
if (semaphore == null) {
    synchronized (serviceKeySemaphoreMap) {
        semaphore = serviceKeySemaphoreMap.get(serviceKey);
        if (semaphore == null) {
            semaphore = new Semaphore(workerThread);
            serviceKeySemaphoreMap.put(serviceKey, semaphore);
        }
    }
}

//获取注册中心服务
IRegisterCenter4Provider registerCenter4Provider =
RegisterCenter.singleton();
List<ProviderService> localProviderCaches =
registerCenter4Provider.getProviderServiceMap().get(serviceKey);
ProviderService localProviderCache = Collections2.filter
(localProviderCaches, new Predicate<ProviderService>() {
    @Override
    public boolean apply(ProviderService input) {
        return StringUtils.equals(input.getServiceMethod().
getName(), methodName);
    }
}).iterator().next();
Object serviceObject = localProviderCache.getServiceObject();

//利用反射发起服务调用
Method method = localProviderCache.getServiceMethod();
Object result = null;
boolean acquire = false;
try {
    //利用 semaphore 实现限流
    acquire = semaphore.tryAcquire(consumeTimeOut,

```

```
TimeUnit.MILLISECONDS);  
  
        if (acquire) {  
  
            //利用反射发起服务调用  
            result = method.invoke(serviceObject, request.getArgs());  
        }  
    } catch (Exception e) {  
        result = e;  
    } finally {  
        if (acquire) {  
            semaphore.release();  
        }  
    }  
  
    //根据服务调用结果组装调用返回对象  
    AresResponse response = new AresResponse();  
    response.setInvokeTimeout(consumeTimeOut);  
    response.setUniqueKey(request.getUniqueKey());  
    response.setResult(result);  
    //将服务调用返回对象回写到消费端  
    ctx.writeAndFlush(response);  
} else {  
    logger.error("-----channel closed!-----");  
}  
}
```

在实现服务端利用反射发起调用服务本地实现 `NettyServerInvokeHandler` 中，为了控制服务端服务能力，使用 `java.util.concurrent.Semaphore` 做了流控处理，具体实现思路如下。

(1) 获取配置信息服务端工作 `workerThread` 线程数 `metaDataModel.getWorkerThreads()`。

(2) 使用 `workerThread` 初始化某个接口的流控基础设施 `Semaphore`，并保存在本地缓存 `Map<String, Semaphore> serviceKeySemaphoreMap = Maps.newConcurrentMap()`中，其

中该缓存的 Key 为服务接口名称。具体代码段为：

```

if (semaphore == null) {
    synchronized (serviceKeySemaphoreMap) {
        semaphore = serviceKeySemaphoreMap.get(serviceKey);
        if (semaphore == null) {
            semaphore = new Semaphore(workerThread);
            serviceKeySemaphoreMap.put(serviceKey, semaphore);
        }
    }
}

```

(3) 在服务接口调用过程中，根据接口名称从本地缓存 serviceKeySemaphoreMap 中获取该服务接口的流控基础设施 semaphore 对象，完成流量控制。具体代码段如下：

```

boolean acquire = false;
try {
    //利用 semaphore 实现限流
    acquire = semaphore.tryAcquire(consumeTimeOut, TimeUnit.MILLISECONDS);
    if (acquire) {
        //利用反射发起服务调用
        result = method.invoke(serviceObject, request.getArgs());
    }
} catch (Exception e) {
    result = e;
} finally {
    if (acquire) {
        semaphore.release();
    }
}

```

至此，我们完成了 Netty 服务端启动、数据序列化编码 Handler、数据反序列化解码 Handler 及服务端服务调用，以及调用结果返回功能的实现。

6.3.2 构建分布式服务框架服务调用端 Netty 客户端

Netty 客户端发起调用，重点需要解决的问题有三个。

- (1) 选择合适的序列化协议，解决 Netty 传输过程中出现的半包/粘包问题。
- (2) 发挥长连接的优势，对 Netty 的 Channel 通道进行复用。
- (3) Netty 是异步框架，客户端发起服务调用后同步等待获取调用结果。

对于问题 (1)，在 6.3.1 节中已有详细说明，在此不再细说。

对于问题 (2)，为使得 Channel 能够复用，编写了一个 Channel 连接池工厂类，针对每一个服务提供者地址，预先生成了一个保存 Channel 的阻塞队列。该 Channel 连接池工厂提供了如下基本操作：

```
/**
 * 初始化 Netty channel 连接队列 Map
 *
 * @param providerMap
 */
public void initChannelPoolFactory(Map<String, List<ProviderService>>
providerMap)

/**
 * 根据服务提供者地址获取对应的 Netty Channel 阻塞队列
 *
 * @param socketAddress
 * @return
 */
public ArrayBlockingQueue<Channel> acquire(InetSocketAddress
socketAddress)
```



```

/**
 * Channel 使用完毕之后, 回收到阻塞队列 arrayBlockingQueue
 *
 * @param arrayBlockingQueue
 * @param channel
 * @param inetSocketAddress
 */
public void release(ArrayBlockingQueue<Channel> arrayBlockingQueue,
Channel channel, InetSocketAddress inetSocketAddress)

/**
 * 为服务提供者地址 socketAddress 注册新的 Channel
 *
 * @param socketAddress
 * @return
 */
public Channel registerChannel(InetSocketAddress socketAddress)

```

下面给出 Channel 连接池工厂类的具体实现。

```

public class NettyChannelPoolFactory {

    private static final Logger logger = LoggerFactory.getLogger(
(NettyChannelPoolFactory.class);

    private static final NettyChannelPoolFactory channelPoolFactory = new
NettyChannelPoolFactory();

    //Key 为服务提供者地址, value 为 Netty Channel 阻塞队列
    private static final Map<InetSocketAddress, ArrayBlockingQueue
<Channel>> channelPoolMap = Maps.newConcurrentMap();

    //初始化 Netty Channel 阻塞队列的长度, 该值为可配置信息

```

```

        private static final int channelConnectSize = PropertyConfigHelper.
getChannelConnectSize();
        //初始化序列化协议类型，该值为可配置信息
        private static final SerializeType serializeType =
PropertyConfigHelper.getSerializeType();
        //服务提供者列表
        private List<ProviderService> serviceMetaDataList =
Lists.newArrayList();

        private NettyChannelPoolFactory() {
        }

        /**
         * 初始化 Netty channel 连接队列 Map
         *
         * @param providerMap
         */
        public void initChannelPoolFactory(Map<String, List<ProviderService>>
providerMap) {
            //将服务提供者信息存入 serviceMetaDataList 列表
            Collection<List<ProviderService>> collectionServiceMetaDataList =
providerMap.values();
            for (List<ProviderService> serviceMetaDataModels :
collectionServiceMetaDataList) {
                if (CollectionUtils.isEmpty(serviceMetaDataModels)) {
                    continue;
                }
                serviceMetaDataList.addAll(serviceMetaDataModels);
            }

            //获取服务提供者地址列表
            Set<InetSocketAddress> socketAddressSet = Sets.newHashSet();
            for (ProviderService serviceMetaData : serviceMetaDataList) {

```

```

String serviceIp = serviceMetaData.getServerIp();
int servicePort = serviceMetaData.getServerPort();

InetSocketAddress socketAddress = new InetSocketAddress
(serviceIp, servicePort);
socketAddressSet.add(socketAddress);
}

//根据服务提供者地址列表初始化 Channel 阻塞队列, 并以地址为 Key, 地址对应的
Channel 阻塞队列为 value, 存入 channelPoolMap
for (InetSocketAddress socketAddress : socketAddressSet) {
    try {
        int realChannelConnectSize = 0;
        while (realChannelConnectSize < channelConnectSize) {
            Channel channel = null;
            while (channel == null) {
                //若 channel 不存在, 则注册新的 Netty Channel
                channel = registerChannel(socketAddress);
            }
            //计数器, 初始化的时候存入阻塞队列的 Netty Channel 个数不超过
            channelConnectSize
            realChannelConnectSize++;

            //将新注册的 Netty Channel 存入阻塞队列 channelArrayBlockingQueue
            //并将阻塞队列 channelArrayBlockingQueue 作为 value 存入
            channelPoolMap
            ArrayBlockingQueue<Channel> channelArrayBlockingQueue =
            channelPoolMap.get(socketAddress);
            if (channelArrayBlockingQueue == null) {
                channelArrayBlockingQueue = new
                ArrayBlockingQueue<Channel>(channelConnectSize);
                channelPoolMap.put(socketAddress,
                channelArrayBlockingQueue);
            }
        }
    }
}

```

```

        }

        channelArrayBlockingQueue.offer(channel);
    }

    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

}

/**
 * 根据服务提供者地址获取对应的 Netty Channel 阻塞队列
 *
 * @param socketAddress
 * @return
 */
public ArrayBlockingQueue<Channel> acquire(InetSocketAddress
socketAddress) {
    return channelPoolMap.get(socketAddress);
}

/**
 * Channel 使用完毕之后，回收到阻塞队列 arrayBlockingQueue
 *
 * @param arrayBlockingQueue
 * @param channel
 * @param inetSocketAddress
 */
public void release(ArrayBlockingQueue<Channel> arrayBlockingQueue,
Channel channel, InetSocketAddress inetSocketAddress) {
    if (arrayBlockingQueue == null) {
        return;
    }
}

```

```

    }

    //回收之前先检查 channel 是否可用,不可用的话,重新注册一个,放入阻塞队列
    if (channel == null || !channel.isActive() || !channel.isOpen()
        || !channel.isWritable()) {
        if (channel != null) {

channel.deregister().syncUninterruptibly().awaitUninterruptibly();

channel.closeFuture().syncUninterruptibly().awaitUninterruptibly();
        }
        Channel newChannel = null;
        while (newChannel == null) {
            logger.debug("-----register new Channel-----");
            newChannel = registerChannel(inetSocketAddress);
        }
        arrayBlockingQueue.offer(newChannel);
        return;
    }
    arrayBlockingQueue.offer(channel);
}

/**
 * 为服务提供者地址 socketAddress 注册新的 Channel
 *
 * @param socketAddress
 * @return
 */
public Channel registerChannel(InetSocketAddress socketAddress) {
    try {
        EventLoopGroup group = new NioEventLoopGroup(10);
        Bootstrap bootstrap = new Bootstrap();
    }

```

```

        bootstrap.remoteAddress(socketAddress);

        bootstrap.group(group)
            .channel(NioSocketChannel.class)
            .option(ChannelOption.TCP_NODELAY, true)
            .handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws
Exception {

                    //注册 Netty 编码器
                    ch.pipeline().addLast(new
NettyEncoderHandler(serializedType));

                    //注册 Netty 解码器
                    ch.pipeline().addLast(new
NettyDecoderHandler(AresResponse.class, serializedType));

                    //注册客户端业务逻辑处理 handler
                    ch.pipeline().addLast(new
NettyClientInvokeHandler());
                }
            });

        ChannelFuture channelFuture = bootstrap.connect().sync();
        final Channel newChannel = channelFuture.channel();
        final CountDownLatch connectedLatch = new CountDownLatch(1);

        final List<Boolean> isSuccessHolder =
Lists.newArrayListWithCapacity(1);
        //监听 Channel 是否建立成功
        channelFuture.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws
Exception {

                //若 Channel 建立成功，保存建立成功的标记

```

```

        if (future.isSuccess()) {
            isSuccessHolder.add(Boolean.TRUE);
        } else {
            //若 Channel 建立失败, 保存建立失败的标记
            future.cause().printStackTrace();
            isSuccessHolder.add(Boolean.FALSE);
        }
        connectedLatch.countDown();
    }
});

connectedLatch.await();
//如果 Channel 建立成功, 返回新建的 Channel
if (isSuccessHolder.get(0)) {
    return newChannel;
}
} catch (Exception e) {
    throw new RuntimeException(e);
}
return null;
}

public static NettyChannelPoolFactory channelPoolFactoryInstance() {
    return channelPoolFactory;
}
}

```

对于问题 (3), Netty 是异步编程框架, 客户端发起请求之后, 不会同步等待结果返回, 需要自己实现同步等待机制。具体实现思路为, 为每次请求新建一个阻塞队列, 返回结果的时候, 存入该阻塞队列, 若在超时时间内返回结果值, 则调用端将该返回结果从阻塞队列中取出返回给调用方, 否则超时, 返回 null。示意图如图 6-17 所示。

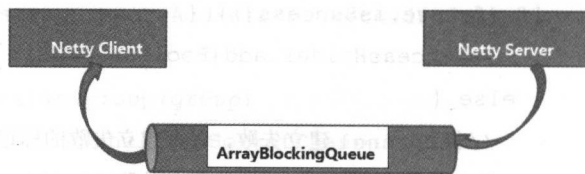


图 6-17 调用端同步使用阻塞队列实现对异步返回结果的同步等待

调用端同步使用阻塞队列实现对异步返回结果的同步等待代码如下。

首先，定义一个 Netty 调用返回结果的包装类 `AresResponseWrapper`，由保存返回结果的阻塞队列 `BlockingQueue<AresResponse>` 与返回时间 `responseTime` 组成。同时定义了判断返回结果是否超时过期的方法 `isExpire()`。用来实现对 Netty 发起异步调用后同步等待功能。

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

/**
 * Netty 异步调用返回结果包装类
 */
public class AresResponseWrapper {
    //存储返回结果的阻塞队列
    private BlockingQueue<AresResponse> responseQueue = new
ArrayBlockingQueue<AresResponse>(1);
    //结果返回时间
    private long responseTime;

    /**
     * 计算该返回结果是否已经过期
     *
     * @return
     */
    public boolean isExpire() {

```



```

AresResponse response = responseQueue.peek();
if (response == null) {
    return false;
}

long timeout = response.getInvokeTimeout();
if ((System.currentTimeMillis() - responseTime) > timeout) {
    return true;
}
return false;
}

public static AresResponseWrapper of() {
    return new AresResponseWrapper();
}

public BlockingQueue<AresResponse> getResponseQueue() {
    return responseQueue;
}

public long getResponseTime() {
    return responseTime;
}

public void setResponseTime(long responseTime) {
    this.responseTime = responseTime;
}
}

```

其次，定义保存及操作返回结果的数据容器类 `RevokerResponseHolder`，主要由以下方法构成：

```
/**
```

```
 * 初始化返回结果容器，requestUniqueKey 唯一标识本次调用
```

```
*
* @param requestUniqueKey
*/
public static void initResponseData(String requestUniqueKey)

/**
* 将 Netty 调用异步返回结果放入阻塞队列
*
* @param response
*/
public static void putResultValue(AresResponse response)

/**
* 从阻塞队列中获取 Netty 异步返回的结果值
*
* @param requestUniqueKey
* @param timeout
* @return
*/
public static AresResponse getValue(String requestUniqueKey, long timeout)
```

具体实现代码细节如下：

```
public class RevokerResponseHolder {
    //服务返回结果 Map
    private static final Map<String, AresResponseWrapper> responseMap =
Maps.newConcurrentMap();
    //清除过期的返回结果
    private static final ExecutorService removeExpireKeyExecutor =
Executors.newSingleThreadExecutor();

    static {
```

//删除超时未获取到结果的Key,防止内存泄漏

```
removeExpireKeyExecutor.execute(new Runnable() {
```

```
    @Override
```

```
    public void run() {
```

```
        while (true) {
```

```
            try {
```

```
                for (Map.Entry<String, AresResponseWrapper> entry :
```

```
responseMap.entrySet()) {
```

```
                    boolean isExpire = entry.getValue().isExpire();
```

```
                    if (isExpire) {
```

```
                        responseMap.remove(entry.getKey());
```

```
                    }
```

```
                    Thread.sleep(10);
```

```
                }
```

```
            } catch (Throwable e) {
```

```
                e.printStackTrace();
```

```
            }
```

```
        }
```

```
    }
```

```
});
```

```
}
```

```
/**
```

```
 * 初始化返回结果容器, requestUniqueKey 唯一标识本次调用
```

```
 *
```

```
 * @param requestUniqueKey
```

```
 */
```

```
public static void initResponseData(String requestUniqueKey) {
```

```
    responseMap.put(requestUniqueKey, AresResponseWrapper.of());
```

```
}
```

```
/**
```

```
 * 将 Netty 调用异步返回结果放入阻塞队列
```

```

    *
    * @param response
    */
    public static void putResultValue(AresResponse response) {
        long currentTime = System.currentTimeMillis();
        AresResponseWrapper responseWrapper =
responseMap.get(response.getUniqueKey());
        responseWrapper.setResponseTime(currentTime);
        responseWrapper.getResponseQueue().add(response);
        responseMap.put(response.getUniqueKey(), responseWrapper);
    }

    /**
    * 从阻塞队列中获取 Netty 异步返回的结果值
    *
    * @param requestUniqueKey
    * @param timeout
    * @return
    */
    public static AresResponse getValue(String requestUniqueKey, long
timeout) {
        AresResponseWrapper responseWrapper = responseMap.get
(requestUniqueKey);
        try {
            return responseWrapper.getResponseQueue().poll(timeout,
TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } finally {
            responseMap.remove(requestUniqueKey);
        }
    }
}

```

至此，我们已经完成了同步等待 Netty 调用结果返回的数据结构定义。我们将在 Channel 连接池工厂类的具体实现 NettyChannelPoolFactory 中实现客户端业务逻辑处理器 NettyClientInvokeHandler。在 NettyClientInvokeHandler 中获取 Netty 异步调用返回的结果，并将该结果保存到 RevokerResponseHolder。

NettyClientInvokeHandler 的具体实现如下：

```
import ares.remoting.framework.model.AresResponse;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;

public class NettyClientInvokeHandler extends SimpleChannelInboundHandler
<AresResponse> {

    public NettyClientInvokeHandler() {

    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
        ctx.flush();
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        cause.printStackTrace();
        ctx.close();
    }

    @Override
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
AresResponse response) throws Exception {
```

```
        //将 Netty 异步返回的结果存入阻塞队列，以便调用端同步获取
        RevokerResponseHolder.putResultValue(response);
    }
}
```

下面继续说明 Netty 客户端如何发起服务调用：

```
/**
 * Netty 请求发起调用线程
 */
public class RevokerServiceCallable implements Callable<AresResponse> {

    private static final Logger logger = LoggerFactory.getLogger(
        (RevokerServiceCallable.class));

    private Channel channel;
    private InetSocketAddress inetSocketAddress;
    private AresRequest request;

    public static RevokerServiceCallable of(InetSocketAddress
        inetSocketAddress, AresRequest request) {
        return new RevokerServiceCallable(inetSocketAddress, request);
    }

    public RevokerServiceCallable(InetSocketAddress inetSocketAddress,
        AresRequest request) {
        this.inetSocketAddress = inetSocketAddress;
        this.request = request;
    }

    @Override
    public AresResponse call() throws Exception {
        //初始化返回结果容器，将本次调用的唯一标识作为 Key 存入返回结果的 Map
    }
}
```

```

RevokerResponseHolder.initResponseData(request.getUniqueKey());
//根据本地调用服务提供者地址获取对应的 Netty 通道 channel 队列
ArrayBlockingQueue<Channel> blockingQueue = NettyChannelPoolFactory.
channelPoolFactoryInstance().acquire(inetSocketAddress);
try {
    if (channel == null) {
        //从队列中获取本次调用的 Netty 通道 channel
        channel = blockingQueue.poll(request.getInvokeTimeout(),
TimeUnit.MILLISECONDS);
    }

    //若获取的 channel 通道已经不可用, 则重新获取一个
    while (!channel.isOpen() || !channel.isActive()
|| !channel.isWritable()) {
        logger.warn("-----retry get new Channel-----");
        channel = blockingQueue.poll(request.getInvokeTimeout(),
TimeUnit.MILLISECONDS);
        if (channel == null) {
            //若队列中没有可用的 Channel, 则重新注册一个 Channel
            channel = NettyChannelPoolFactory.
channelPoolFactoryInstance().registerChannel(inetSocketAddress);
        }
    }

    //将本次调用的信息写入 Netty 通道, 发起异步调用
ChannelFuture channelFuture = channel.writeAndFlush(request);
channelFuture.syncUninterruptibly();

    //从返回结果容器中获取返回结果, 同时设置等待超时时间为 invokeTimeout
    long invokeTimeout = request.getInvokeTimeout();
    return RevokerResponseHolder.getValue(request.getUniqueKey(),
invokeTimeout);
} catch (Exception e) {

```

```
        logger.error("service invoke error.", e);
    } finally {
        //本次调用完毕后，将Netty的通道channel重新释放到队列中，以便下次调用复用
        NettyChannelPoolFactory.channelPoolFactoryInstance().release
        (blockingQueue, channel, inetSocketAddress);
    }
    return null;
}
}
```

总结一下，在分布式服务框架调用端通过 Netty 客户端发起一次调用，并得到调用返回结果，整个过程步骤如下。

- (1) 获取服务提供者列表，通过某种软负载算法选择某一个服务提供者。
- (2) 根据服务提供者信息从 Netty 连接池中获取对应的 Channel 连接。
- (3) 将服务请求数据对象通过某种序列化协议编码成字节数组，通过通道 Channel 发送到服务端。
- (4) 同步等待服务端返回调用结果，最终完成一次服务调用。

步骤 (1) 及软负载将在第 7 章讲述。

步骤 (2) 的具体实现代码段如下：

```
//根据本地调用服务提供者地址获取对应的Netty通道channel队列
ArrayBlockingQueue<Channel> blockingQueue = NettyChannelPoolFactory.
channelPoolFactoryInstance().acquire(inetSocketAddress);
try {
    if (channel == null) {
        //从队列中获取本次调用的Netty通道channel
        channel = blockingQueue.poll(request.getInvokeTimeout(),
        TimeUnit.MILLISECONDS);
    }
}
```


.....省略.....

步骤（3）在初始化 Netty Channel 连接池的时候实现，代码段如下。

在 NettyChannelPoolFactory 类中：

```
.....省略.....
//注册 Netty 编码器
ch.pipeline().addLast(new NettyEncoderHandler(serializedType));
//注册 Netty 解码器
ch.pipeline().addLast(new NettyDecoderHandler(AresResponse.class,
serializedType));
.....省略.....
```

其中 NettyEncoderHandler 与 NettyDecoderHandler 的实现在 6.3.1 节中已有详细讲解。

步骤（4）的实现代码段如下：

```
.....省略.....
//从返回结果容器中获取返回结果，同时设置等待超时时间为 invokeTimeout
long invokeTimeout = request.getInvokeTimeout();
return RevokerResponseHolder.getValue(request.getUniqueKey(),
invokeTimeout);
.....省略.....
```

6.4 本章小结

本章介绍了 Java I/O 体系的知识图谱，并进一步介绍了 Netty 的常用使用方法。最后介绍了如何使用 Netty 来构建分布式服务框架服务提供端及客户发起端，并针对每一个知识点给出了具体的代码实现。

第 7 章

分布式服务框架软负载实现

7.1 软负载的实现原理

负载均衡的目的是将请求按照某种策略分布到多台机器上，使得系统能够实现横向扩展，是应用实现可伸缩性的关键技术，也是系统能够应对大流量的核心技术之一。分布式服务框架中实现负载均衡是通过软件算法来实现的，有别于基于硬件设备（如 F5）实现负载均衡，故称为软负载。

在分布式服务框架中，负载均衡是在服务消费端实现的，其实现原理如下。

- ◎ 服务消费端在应用启动之初从服务注册中心获取服务提供者列表，缓存到服务调用端本地缓存。
- ◎ 服务消费端发起服务调用之前，先通过某种策略或者算法从服务提供者列表本地缓存中选择本次调用的目标机器，再发起服务调用，从而完成负载均衡的功能。

7.2 负载均衡常用算法

负载均衡常用算法主要有随机、加权随机、轮询、加权轮询、源地址 hash 等。

7.2.1 软负载随机算法实现

随机算法原理为：获取服务列表大小范围内的随机数，将该随机数作为列表索引，从服务提供列表中获取服务提供者。

为负载均衡策略算法定义接口如下：

```
public interface ClusterStrategy {
    public ProviderService select(List<ProviderService>
providerServices);
}
```

其中接口方法入参 `providerServices` 为服务提供者列表。

软负载随机算法的实现过程如下。

```
import ares.remoting.framework.cluster.ClusterStrategy;
import ares.remoting.framework.model.ProviderService;
import org.apache.commons.lang3.RandomUtils;
import java.util.List;

public class RandomClusterStrategyImpl implements ClusterStrategy {
    @Override
    public ProviderService select(List<ProviderService> providerServices) {
        int MAX_LEN = providerServices.size();
        int index = RandomUtils.nextInt(0, MAX_LEN - 1);
        return providerServices.get(index);
    }
}
```

```
}  
}
```

实现原理：获得服务提供者列表大小区间之间的随机数，作为服务提供者列表的索引来获取服务。

7.2.2 软负载加权随机算法实现

加权随机算法在随机算法的基础上针对权重做了处理。软负载加权随机算法实现代码如下。

```
import ares.remoting.framework.cluster.ClusterStrategy;  
import ares.remoting.framework.model.ProviderService;  
import com.google.common.collect.Lists;  
import org.apache.commons.lang3.RandomUtils;  
  
import java.util.List;  
  
public class WeightRandomClusterStrategyImpl implements ClusterStrategy {  
  
    @Override  
    public ProviderService select(List<ProviderService> providerServices) {  
        //存放加权后的服务提供者列表  
        List<ProviderService> providerList = Lists.newArrayList();  
        for (ProviderService provider : providerServices) {  
            int weight = provider.getWeight();  
            for (int i = 0; i < weight; i++) {  
                providerList.add(provider.copy());  
            }  
        }  
  
        int MAX_LEN = providerList.size();
```

```

        int index = RandomUtils.nextInt(0, MAX_LEN - 1);
        return providerList.get(index);
    }
}

```

实现原理：首先根据加权数放大服务提供者列表，比如服务提供者 A 加权数为 3，放大之后变为 A, A, A，存放在新的服务提供者列表，然后对新的服务提供者列表应用随机算法。

7.2.3 软负载轮询算法实现

轮询算法，将服务调用请求按顺序轮流分配到服务提供者后端服务器上，均衡对待每一台服务提供者机器。软负载轮询算法实现代码如下。

```

import ares.remoting.framework.cluster.ClusterStrategy;
import ares.remoting.framework.model.ProviderService;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PollingClusterStrategyImpl implements ClusterStrategy {
    //计数器
    private int index = 0;
    private Lock lock = new ReentrantLock();

    @Override
    public ProviderService select(List<ProviderService> providerServices) {

        ProviderService service = null;
        try {
            lock.tryLock(10, TimeUnit.MILLISECONDS);

```

```

        //若计数大于服务提供者个数，将计数器归 0
        if (index >= providerServices.size()) {
            index = 0;
        }
        service = providerServices.get(index);
        index++;
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
    //兜底，保证程序健壮性，若未取到服务，则直接取第 1 个
    if (service == null) {
        service = providerServices.get(0);
    }
    return service;
}
}

```

实现原理：依次按顺序获取服务提供者列表中的数据，并使用计数器记录使用过的数据索引，若数据索引到最后一个数据，则计数器归零，重新开始新的循环。

7.2.4 软负载加权轮询算法实现

加权轮询算法是在轮询算法的基础上对权重做了处理。软负载加权轮询算法实现代码如下。

```

import ares.remoting.framework.cluster.ClusterStrategy;
import ares.remoting.framework.model.ProviderService;
import com.google.common.collect.Lists;

```

```
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class WeightPollingClusterStrategyImpl implements ClusterStrategy {
    //计数器
    private int index = 0;
    private Lock lock = new ReentrantLock();

    @Override
    public ProviderService select(List<ProviderService> providerServices) {
        ProviderService service = null;
        try {
            lock.tryLock(10, TimeUnit.MILLISECONDS);
            //存放加权后的服务提供者列表
            List<ProviderService> providerList = Lists.newArrayList();
            for (ProviderService provider : providerServices) {
                int weight = provider.getWeight();
                for (int i = 0; i < weight; i++) {
                    providerList.add(provider.copy());
                }
            }
            //若计数大于服务提供者个数，将计数器归0
            if (index >= providerList.size()) {
                index = 0;
            }
            service = providerList.get(index);
            index++;
            return service;
        } catch (InterruptedException e) {
```

```
        e.printStackTrace();
    } finally {
        lock.unlock();
    }

    //兜底，保证程序健壮性，若未取到服务，则直接取第 1 个
    return providerServices.get(0);
}
}
```

实现原理：首先根据加权数放大服务提供者列表，再在放大后的服务提供者基础上使用轮询算法获取服务提供者。

7.2.5 软负载源地址 hash 算法实现

源地址 hash 算法实际是利于请求来源的 IP 的 hashCode 对服务提供者列表大小取模，得到服务提供者列表索引，进而获取到服务提供者。软负载源地址 hash 算法实现代码如下。

```
import ares.remoting.framework.Helper.IPHelper;
import ares.remoting.framework.cluster.ClusterStrategy;
import ares.remoting.framework.model.ProviderService;
import java.util.List;

public class HashClusterStrategyImpl implements ClusterStrategy {

    @Override
    public ProviderService select(List<ProviderService> providerServices) {
        //获取调用方 IP
        String localIP = IPHelper.localIp();
        //获取源地址对应的 hashCode
        int hashCode = localIP.hashCode();
```



```

//获取服务列表大小
int size = providerServices.size();

return providerServices.get(hashCode % size);
}
}

```

实现原理：使用调用方 IP 地址的 hash 值，将服务列表大小取模后的值作为服务列表索引，根据该索引取值。

7.3 实现自己的软负载机制

我们需要将以上算法实现整合到我们的分布式服务框架的实现中去。为此定义一个软负载策略引擎类。使用门面模式，对外暴露统一简单的 API 界面，根据不同的策略配置来选取不同的策略服务。策略引擎实现具体代码如下。

```

1  import ares.remoting.framework.cluster.ClusterStrategy;
2  import ares.remoting.framework.cluster.impl.*;
3  import com.google.common.collect.Maps;
4  import java.util.Map;
5
6  public class ClusterEngine {
7
8      private static final Map<ClusterStrategyEnum, ClusterStrategy>
9      clusterStrategyMap = Maps.newConcurrentMap();
10
11     static {
12         clusterStrategyMap.put(ClusterStrategyEnum.Random, new
13 RandomClusterStrategyImpl());
14         clusterStrategyMap.put(ClusterStrategyEnum.WeightRandom, new
15 WeightRandomClusterStrategyImpl());

```

```
16         clusterStrategyMap.put(ClusterStrategyEnum.Polling, new
17 PollingClusterStrategyImpl());
18         clusterStrategyMap.put(ClusterStrategyEnum.WeightPolling, new
19 WeightPollingClusterStrategyImpl());
20         clusterStrategyMap.put(ClusterStrategyEnum.Hash, new
21 HashClusterStrategyImpl());
22     }
23
24
25     public static ClusterStrategy queryClusterStrategy(String
clusterStrategy) {
26         ClusterStrategyEnum clusterStrategyEnum =
27 ClusterStrategyEnum.queryByCode(clusterStrategy);
28         if (clusterStrategyEnum == null) {
29             //默认选择随机算法
30             return new RandomClusterStrategyImpl();
31         }
32         return clusterStrategyMap.get(clusterStrategyEnum);
33     }
34 }
```

第 11~22 行，在静态代码块中将软负载算法实现注册到 `clusterStrategyMap` 中，key 为代表算法的枚举字符串，value 为该算法的具体实现 Bean。

第 25~35 行，对外暴露统一的静态方法 API，根据传入的枚举字符串，从 `clusterStrategyMap` 中获取对应的软负载算法实现。

其中枚举 `ClusterStrategyEnum` 定义了软负载策略类别：

```
public enum ClusterStrategyEnum {
    //随机算法
    Random("Random"),
    //权重随机算法
    WeightRandom("WeightRandom"),
```

```

//轮询算法
Polling("Polling"),
//权重轮询算法
WeightPolling("WeightPolling"),
//源地址 hash 算法
Hash("Hash");

```

.....省略.....

```

}

```

7.4 软负载在分布式服务框架中的应用

第6章讲述了服务客户端调用过程的第(1)步：“获取服务提供者列表，通过某种软负载算法选择某一个服务提供者”。下面介绍在分布式服务框架中客户端在发起调用的时候，如何使用软负载算法，选择某一个服务提供方发起调用，代码如下。

```

1  public class RevokerProxyBeanFactory implements InvocationHandler {
2
3      private ExecutorService fixedThreadPool = null;
4      //服务接口
5      private Class<?> targetInterface;
6      //超时时间
7      private int consumeTimeout;
8      //调用者线程数
9      private static int threadWorkerNumber = 10;
10     //负载均衡策略
11     private String clusterStrategy;
12
13
14     public RevokerProxyBeanFactory(Class<?> targetInterface, int
15     consumeTimeout, String clusterStrategy) {

```

```
16         this.targetInterface = targetInterface;
17         this.consumeTimeout = consumeTimeout;
18         this.clusterStrategy = clusterStrategy;
19     }
20
21     @Override
22     public Object invoke(Object proxy, Method method, Object[] args)
23     throws Throwable {
24         //服务接口名称
25         String serviceKey = targetInterface.getName();
26         //获取某个接口的服务提供者列表
27         IRegisterCenter4Invoker registerCenter4Consumer =
RegisterCenter.singleton();
28         List<ProviderService> providerServices = registerCenter4Consumer.
getServiceMetaDataMap4Consume().get(serviceKey);
29
30
31         //根据客户端配置信息选择软负载策略具体实现
32         ClusterStrategy clusterStrategyService =
33         ClusterEngine.queryClusterStrategy(clusterStrategy);
34         //根据软负载策略，从服务提供者列表选取本次调用的服务提供者
35         ProviderService providerService =
36         clusterStrategyService.select(providerServices);
37
38         //复制一份服务提供者信息
39         ProviderService newProvider = providerService.copy();
40         //设置本次调用服务的方法及接口
41         newProvider.setServiceMethod(method);
42         newProvider.setServiceItf(targetInterface);
43
44         //声明调用 AresRequest 对象，AresRequest 表示发起一次调用所包含的信息
45         final AresRequest request = new AresRequest();
46         //设置本次调用的唯一标识
47         request.setUniqueKey(UUID.randomUUID().toString() + "-" +
```

```

48 Thread.currentThread().getId());
49     //设置本次调用的服务提供者信息
50     request.setProviderService(newProvider);
51     //设置本次调用的超时时间
52     request.setInvokeTimeout(consumeTimeout);
53     //设置本次调用的方法名称
54     request.setInvokedMethodName(method.getName());
55     //设置本次调用的方法参数信息
56     request.setArgs(args);
57
58     try {
59         //构建用来发起调用的线程池
60         if (fixedThreadPool == null) {
61             synchronized (RevokerProxyBeanFactory.class) {
62                 if (null == fixedThreadPool) {
63                     fixedThreadPool = Executors.newFixedThreadPool
64 (threadWorkerNumber);
65                 }
66             }
67         }
68         //根据服务提供者的 IP、port, 构建 InetSocketAddress 对象, 标识服务
提供者地址
69         String serverIp =
request.getProviderService().getServerIp();
70         int serverPort =
request.getProviderService().getServerPort();
71         InetSocketAddress inetSocketAddress = new
72 InetSocketAddress(serverIp, serverPort);
73
74 //提交本次调用信息到线程池 fixedThreadPool, 发起调用
75         Future<AresResponse> responseFuture = fixedThreadPool.submit
76 (RevokerServiceCallable.of(inetSocketAddress, request));
77         //获取调用的返回结果

```

```

78         AresResponse response = responseFuture.get(request.
79 getInvokeTimeout(), TimeUnit.MILLISECONDS);
80         if (response != null) {
81             return response.getResult();
82         }
83     } catch (Exception e) {
84         throw new RuntimeException(e);
85     }
86     return null;
87 }
88
89     public Object getProxy() {
90         return Proxy.newProxyInstance(Thread.currentThread().
91 getContextClassLoader(), new Class<?>[]{targetInterface}, this);
92     }
93     private static volatile RevokerProxyBeanFactory singleton;
94
95     public static RevokerProxyBeanFactory singleton(Class<?>
96 targetInterface, int consumeTimeout, String clusterStrategy) throws
Exception {
97         if (null == singleton) {
98             synchronized (RevokerProxyBeanFactory.class) {
99                 if (null == singleton) {
100                     singleton = new RevokerProxyBeanFactory
101 (targetInterface, consumeTimeout, clusterStrategy);
102                 }
103             }
104         }
105         return singleton;
106     }
107 }

```

整个类 `RevokerProxyBeanFactory` 是远程服务在服务调用方的动态代理类实现，通过

实现 `InvocationHandler` 接口，将复杂的远程调用通信逻辑封装在方法 `public Object invoke(Object proxy, Method method, Object[] args)` 内部。

整个远程通信调用逻辑如下。

- ◎ 第 25~30 行，从服务注册中心获取服务提供者列表。
- ◎ 第 32~37 行是实现软负载的关键代码，根据服务调用端配置的软负载均衡策略参数 `clusterStrategy`，作为方法 `ClusterEngine.queryClusterStrategy()` 的参数，获取到算法具体实现 `clusterStrategyService`，调用方法 `clusterStrategyService.select()` 即可获得某个服务提供者。
- ◎ 第 45~57 行，组装服务调用请求 `AresRequest` 对象。
- ◎ 第 76~79 行，异步提交调用请求。
- ◎ 第 81~89 行，通过阻塞队列机制同步等待请求返回结果。

7.5 本章小结

本章介绍了多种软负载实现算法，并给出了实际实现代码。同时讲解了在分布式服务框架中如何结合负载算法实现软负载调度。

第 8 章

分布式服务框架服务治理

8.1 服务治理介绍

在大规模服务化之前，系统应用之间的交互可能只是简单地通过 `WebService`、`RMI` 等 `RPC` 框架来实现，通过手工配置调用端服务地址进行调用，通过 `F5` 等硬件进行负载均衡。但是随着业务的不断演进，服务个数越来越多的时候，这种做法就遇到了瓶颈，因为服务数多本身就是问题，量变引发质变，服务数多了，会导致很多问题。

问题一：随着服务的增多，服务之间的依赖关系变得越来越复杂，靠人力很难梳理清楚整个链路服务之间的依赖关系。这样会导致很多风险，比如系统发布需要先发布下游服务，再发布上游服务，若无法梳理清楚服务之间的依赖，就无法正确安排系统发布顺序。此时，需要依赖服务治理功能，自动画出应用之间的依赖关系图。

问题二：需要对每个服务本身的服务质量了如指掌才能保证整个链路服务的稳定性。服务质量包括服务 `QPS`、每天调用总量、`top50`、`top90`、`top99` 响应时间等指标。

问题三：随着服务数量的增多，沟通成本随之增加，需要对每个服务标注负责人。

问题四：当发现某个非关键服务出错率很高，对业务关键链路造成了影响，要有一键降级的功能将该服务从调用链路中摘除。

问题五：对某个已有服务升级之后，需要在线上环境进行灰度发布或者 AB 测试。要求服务有自动分组能力，某个消费组的请求只打到对应的服务组上。

问题六：服务部署集群中，可能某些机器配置更好，有更高的服务吞吐能力，能支持更高的 QPS，要求可以对该机器配置更高的服务权重，使得请求按权重比例打到该机器上。

问题七：当调用链路横跨多个服务、多个应用，对每一次调用需要有一个唯一标识将服务之间的调用串联起来，有助于排查线上问题。

服务治理是一个很大的主题，它涵盖了非常多的内容。限于篇幅，无法做到面面俱到，这里只对相关的部分内容做一个概要性质的介绍，可以将部分内容归纳为以下主题。

- ◎ 服务注册与发现。
- ◎ 软负载。
- ◎ 服务质量监控与服务指标数据采集。
- ◎ 记录负责人。
- ◎ 服务分组路由。
- ◎ 服务依赖关系分析。
- ◎ 服务降级。
- ◎ 服务权重调整。
- ◎ 服务调用链路跟踪。

8.2 服务治理的简单实现

因为篇幅关系，我们将选取实现以上的部分服务治理功能，包括服务注册与发现、软负载、服务分组路由、服务依赖关系分析、服务权重调整、服务调用链路跟踪。其中服务注册与发现、软负载、服务权重调整在之前的章节已经有所介绍，在此不再叙述。

8.2.1 服务分组路由实现

服务分组是指，在服务提供者集群中，将集群分为两组机器，机器组 A 与机器组 B，在消费者集群中，可以指定某个消费者调用某个集群，比如指定某台机器的请求全部打到机器组 A 上面，这样做的好处是可以实现灰度发布或 AB 测试功能。

服务分组的实现原理：在第 5 章图 5-6 中，注册中心的路径为“根目录/APP_KEY/服务类路径/（服务提供者或者消费者类型）/（IP、端口等信息数据）”。要实现服务分组，可以在“APP_KEY/服务类路径”中间再分一层服务组名。加入服务组名之后，注册中心的路径变为“根目录/APP_KEY/服务组名/服务类路径/（服务提供者或者消费者类型）/（IP、端口等信息数据）”。注册中心加入服务组名路径之后，指定消费某个服务组的消费端将该服务组下的服务提供者列表获取到本地缓存，消费端服务调用的时候，将按照指定的软负载算法从本地缓存中选取一个服务调用者发起调用。这样就达到了服务分组调用的目的，如图 8-1 所示。

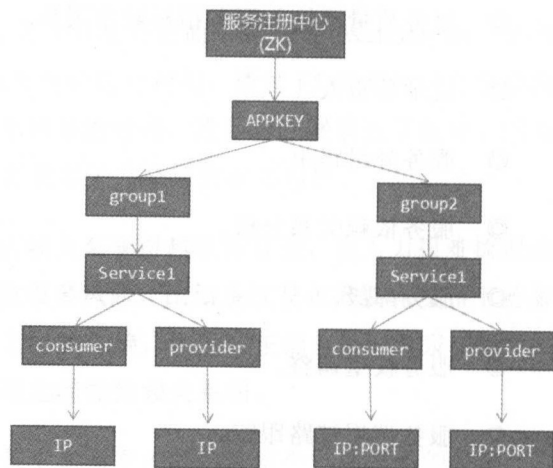


图 8-1 注册中心服务分组示意图

加入分组机制之后，注册中心的实现类代码如下。

```
public class RegisterCenter implements IRegisterCenter4Invoker,
IRegisterCenter4Provider {

    private static RegisterCenter registerCenter = new RegisterCenter();

    //服务提供者列表, Key:服务提供者接口 value:服务提供者服务方法列表
    private static final Map<String, List<ProviderService>>
providerServiceMap = Maps.newConcurrentMap();

    //服务端 ZK 服务元信息, 选择服务 (第一次直接从 ZK 拉取, 后续由 ZK 的监听机制主动更新)
    private static final Map<String, List<ProviderService>>
serviceMetaDataMap4Consume =
com.google.common.collect.Maps.newConcurrentMap();

    private static String ZK_SERVICE = PropertyConfigHelper.
getZkService();

    private static int ZK_SESSION_TIME_OUT = PropertyConfigHelper.
getZkConnectionTimeout();

    private static int ZK_CONNECTION_TIME_OUT = PropertyConfigHelper.
getZkConnectionTimeout();

    private static String ROOT_PATH = "/config_register";
    public static String PROVIDER_TYPE = "provider";
    public static String INVOKER_TYPE = "consumer";
    private static volatile ZkClient zkClient = null;

    private RegisterCenter() {
    }

    public static RegisterCenter singleton() {
        return registerCenter;
    }
}
```

```
@Override
public void registerProvider(final List<ProviderService>
serviceMetaData) {
    if (CollectionUtils.isEmpty(serviceMetaData)) {
        return;
    }

    //连接 ZK, 注册服务
    synchronized (RegisterCenter.class) {
        for (ProviderService provider : serviceMetaData) {
            String serviceItfKey = provider.getServiceItf().getName();

            List<ProviderService> providers =
providerServiceMap.get(serviceItfKey);
            if (providers == null) {
                providers = Lists.newArrayList();
            }
            providers.add(provider);
            providerServiceMap.put(serviceItfKey, providers);
        }

        if (zkClient == null) {
            zkClient = new ZkClient(ZK_SERVICE, ZK_SESSION_TIME_OUT,
ZK_CONNECTION_TIME_OUT, new SerializableSerializer());
        }

        //创建 ZK 命名空间/当前部署应用 APP 命名空间
        String APP_KEY = serviceMetaData.get(0).getAppKey();
        String ZK_PATH = ROOT_PATH + "/" + APP_KEY;
        boolean exist = zkClient.exists(ZK_PATH);
        if (!exist) {
            zkClient.createPersistent(ZK_PATH, true);
        }
    }
}
```

```

    }

    for (Map.Entry<String, List<ProviderService>> entry :
providerServiceMap.entrySet()) {
        //服务分组
        String groupName = entry.getValue().get(0).getGroupName();
        //创建服务提供者
        String serviceNode = entry.getKey();
        String servicePath = ZK_PATH + "/" + groupName + "/" +
serviceNode + "/" + PROVIDER_TYPE;
        exist = zkClient.exists(servicePath);
        if (!exist) {
            zkClient.createPersistent(servicePath, true);
        }

        //创建当前服务器节点
        int serverPort = entry.getValue().get(0).getServerPort();//
服务端口

        int weight = entry.getValue().get(0).getWeight();//服务权重
        int workerThreads = entry.getValue().get(0).
getWorkerThreads();//服务工作线程
        String localIp = IPHelper.localIp();
        String currentServiceIpNode = servicePath + "/" + localIp +
"|" + serverPort + "|" + weight + "|" + workerThreads + "|" + groupName;
        exist = zkClient.exists(currentServiceIpNode);
        if (!exist) {
            //注意，这里创建的是临时节点
            zkClient.createEphemeral(currentServiceIpNode);
        }

        //监听注册服务的变化，同时更新数据到本地缓存
        zkClient.subscribeChildChanges(servicePath, new
IZkChildListener() {

```

```
        @Override
        public void handleChildChange(String parentPath,
List<String> currentChilds) throws Exception {
            if (currentChilds == null) {
                currentChilds = Lists.newArrayList();
            }

            //存活的服务 IP 列表
            List<String> activityServiceIpList =
Lists.newArrayList(Lists.transform(currentChilds, new Function<String,
String>() {

                @Override
                public String apply(String input) {
                    return StringUtils.split(input, "|")[0];
                }
            }));
            refreshActivityService(activityServiceIpList);
        }
    });
}

}

@Override
public Map<String, List<ProviderService>> getProviderServiceMap() {
    return providerServiceMap;
}

@Override
public void initProviderMap(String remoteAppKey, String groupName) {
    if (MapUtils.isEmpty(serviceMetaDataMap4Consume)) {
        serviceMetaDataMap4Consume.putAll(fetchOrUpdateServiceMetaData
```

```

(remoteAppKey, groupName));
    }
}

@Override
public Map<String, List<ProviderService>> getServiceMetaDataMap4
Consume() {
    return serviceMetaDataMap4Consume;
}

@Override
public void registerInvoker(InvokerService invoker) {
    if (invoker == null) {
        return;
    }

    //连接 ZK, 注册服务
    synchronized (RegisterCenter.class) {

        if (zkClient == null) {
            zkClient = new ZkClient(ZK_SERVICE, ZK_SESSION_TIME_OUT,
ZK_CONNECTION_TIME_OUT, new SerializableSerializer());
        }

        //创建 ZK 命名空间/当前部署应用 APP 命名空间
        boolean exist = zkClient.exists(ROOT_PATH);
        if (!exist) {
            zkClient.createPersistent(ROOT_PATH, true);
        }

        //创建服务消费者节点
        String remoteAppKey = invoker.getRemoteAppKey();
        String groupName = invoker.getGroupName();
    }
}

```

```
String serviceNode = invoker.getServiceItf().getName();
String servicePath = ROOT_PATH + "/" + remoteAppKey + "/" +
groupName + "/" + serviceNode + "/" + INVOKER_TYPE;
exist = zkClient.exists(servicePath);
if (!exist) {
    zkClient.createPersistent(servicePath, true);
}

//创建当前服务器节点
String localIp = IPHelper.localIp();
String currentServiceIpNode = servicePath + "/" + localIp;
exist = zkClient.exists(currentServiceIpNode);
if (!exist) {
    //注意，这里创建的是临时节点
    zkClient.createEphemeral(currentServiceIpNode);
}
}

//利用 ZK 自动刷新当前存活的服务提供者列表数据
private void refreshActivityService(List<String> serviceIpList) {
    if (serviceIpList == null) {
        serviceIpList = Lists.newArrayList();
    }

    Map<String, List<ProviderService>> currentServiceMetaDataMap =
Maps.newHashMap();
    for (Map.Entry<String, List<ProviderService>> entry :
providerServiceMap.entrySet()) {
        String key = entry.getKey();
        List<ProviderService> providerServices = entry.getValue();

        List<ProviderService> serviceMetaDataModelList =
```



```

currentServiceMetaDataMap.get(key);

    if (serviceMetaDataModelList == null) {
        serviceMetaDataModelList = Lists.newArrayList();
    }

    for (ProviderService serviceMetaData : providerServices) {
        if (serviceIpList.contains(serviceMetaData.getServerIp())) {
            serviceMetaDataModelList.add(serviceMetaData);
        }
    }

    currentServiceMetaDataMap.put(key, serviceMetaDataModelList);
}

providerServiceMap.clear();
providerServiceMap.putAll(currentServiceMetaDataMap);
}

private void refreshServiceMetaDataMap(List<String> serviceIpList) {
    if (serviceIpList == null) {
        serviceIpList = Lists.newArrayList();
    }

    Map<String, List<ProviderService>> currentServiceMetaDataMap =
Maps.newHashMap();

    for (Map.Entry<String, List<ProviderService>> entry :
serviceMetaDataMap4Consume.entrySet()) {
        String serviceItfKey = entry.getKey();
        List<ProviderService> serviceList = entry.getValue();

        List<ProviderService> providerServiceList =
currentServiceMetaDataMap.get(serviceItfKey);

        if (providerServiceList == null) {
            providerServiceList = Lists.newArrayList();

```

```
    }

    for (ProviderService serviceMetaData : serviceList) {
        if (serviceIpList.contains(serviceMetaData.getServerIp())) {
            providerServiceList.add(serviceMetaData);
        }
    }

    currentServiceMetaDataMap.put(serviceItfKey,
providerServiceList);
    }

    serviceMetaDataMap4Consume.clear();
    serviceMetaDataMap4Consume.putAll(currentServiceMetaDataMap);
}

private Map<String, List<ProviderService>>
fetchOrUpdateServiceMetaData(String remoteAppKey, String groupName) {
    final Map<String, List<ProviderService>> providerServiceMap =
Maps.newConcurrentMap();

    //连接 ZK
    synchronized (RegisterCenter.class) {
        if (zkClient == null) {
            zkClient = new ZkClient(ZK_SERVICE, ZK_SESSION_TIME_OUT,
ZK_CONNECTION_TIME_OUT, new SerializableSerializer());
        }
    }

    //从 ZK 获取服务提供者列表
    String providePath = ROOT_PATH + "/" + remoteAppKey + "/" + groupName;
    List<String> providerServices = zkClient.getChildren(providePath);

    for (String serviceName : providerServices) {
```

```

String servicePath = providePath + "/" + serviceName + "/" +
PROVIDER_TYPE;
List<String> ipPathList = zkClient.getChildren(servicePath);
for (String ipPath : ipPathList) {
    String serverIp = StringUtils.split(ipPath, "|")[0];
    String serverPort = StringUtils.split(ipPath, "|")[1];
    int weight = Integer.parseInt(StringUtils.split(ipPath,
"|")[2]);

    int workerThreads =
Integer.parseInt(StringUtils.split(ipPath, "|")[3]);
    String group = StringUtils.split(ipPath, "|")[4];

    List<ProviderService> providerServiceList =
providerServiceMap.get(serviceName);
    if (providerServiceList == null) {
        providerServiceList = Lists.newArrayList();
    }
    ProviderService providerService = new ProviderService();

    try {
        providerService.setServiceItf(ClassUtils.getClass
(serviceName));
    } catch (ClassNotFoundException e) {
        throw new RuntimeException(e);
    }

    providerService.setServerIp(serverIp);
    providerService.setServerPort(Integer.parseInt
(serverPort));

    providerService.setWeight(weight);
    providerService.setWorkerThreads(workerThreads);
    providerService.setGroupName(group);
    providerServiceList.add(providerService);
}

```

```

        providerServiceMap.put(serviceName, providerServiceList);
    }

    //监听注册服务的变化,同时更新数据到本地缓存
    zkClient.subscribeChildChanges(servicePath, new
    IZkChildListener() {
        @Override
        public void handleChildChange(String parentPath, List<String>
currentChilds) throws Exception {
            if (currentChilds == null) {
                currentChilds = Lists.newArrayList();
            }
            currentChilds = Lists.newArrayList(Lists.transform
(currentChilds, new Function<String, String>() {
                @Override
                public String apply(String input) {
                    return StringUtils.split(input, "|")[0];
                }
            }));
            refreshServiceMetaDataMap(currentChilds);
        }
    });
    return providerServiceMap;
}
}

```

8.2.2 简单服务依赖关系分析实现

服务依赖关系分析,限于篇幅这里只实现最简单的服务依赖分析,可以实时获得服务提供者信息与对应的服务消费者信息。

实现原理：服务提供者信息与对应的服务消费者信息在注册中心已经存在了，所需要做的，不过是提供获取服务提供者信息与消费者信息列表的接口方法，从注册中心查找对应的信息。

定义服务治理接口 `IRegisterCenter4Governance`：

```
import ares.remoting.framework.model.InvokerService;
import ares.remoting.framework.model.ProviderService;
import org.apache.commons.lang3.tuple.Pair;
import java.util.List;

public interface IRegisterCenter4Governance {

    /**
     * 获取服务提供者列表与服务消费者列表
     *
     * @param serviceName
     * @param appKey
     * @return
     */
    public Pair<List<ProviderService>, List<InvokerService>> queryProvidersAndInvokers(String serviceName, String appKey);

}
```

接口的具体实现如下：

```
public class RegisterCenter implements IRegisterCenter4Invoker,
IRegisterCenter4Provider, IRegisterCenter4Governance {

    private static RegisterCenter registerCenter = new RegisterCenter();

    //服务提供者列表, Key: 服务提供者接口 value: 服务提供者服务方法列表
```

```
private static final Map<String, List<ProviderService>>
providerServiceMap = Maps.newConcurrentMap();

//服务端 zk 服务元信息，选择服务（第一次直接从 zk 拉取，后续由 zk 的监听机制主动更新）
private static final Map<String, List<ProviderService>>
serviceMetaDataMap4Consume
=
com.google.common.collect.Maps.newConcurrentMap();

private static String ZK_SERVICE = PropertyConfigHelper.
getZkService();

private static int ZK_SESSION_TIME_OUT = PropertyConfigHelper.
getZkConnectionTimeout();

private static int ZK_CONNECTION_TIME_OUT = PropertyConfigHelper.
getZkConnectionTimeout();

private static String ROOT_PATH = "/config_register";
public static String PROVIDER_TYPE = "provider";
public static String INVOKER_TYPE = "consumer";
private static volatile ZkClient zkClient = null;

private RegisterCenter() {
}

public static RegisterCenter singleton() {
    return registerCenter;
}

.....省略.....

@Override
public Pair<List<ProviderService>, List<InvokerService>>
queryProvidersAndInvokers(String serviceName, String appKey) {
    //服务消费者列表
    List<InvokerService> invokerServices = Lists.newArrayList();
```

```

//服务提供者列表
List<ProviderService> providerServices = Lists.newArrayList();

//连接 ZK
if (zkClient == null) {
    synchronized (RegisterCenter.class) {
        if (zkClient == null) {
            zkClient = new ZkClient(ZK_SERVICE, ZK_SESSION_TIME_OUT,
ZK_CONNECTION_TIME_OUT, new SerializableSerializer());
        }
    }
}

String parentPath = ROOT_PATH + "/" + appKey;
//获取 ROOT_PATH + APP_KEY 注册中心子目录列表
List<String> groupServiceList = zkClient.getChildren(parentPath);
if (CollectionUtils.isEmpty(groupServiceList)) {
    return Pair.of(providerServices, invokerServices);
}

for (String group : groupServiceList) {
    String groupPath = parentPath + "/" + group;
    //获取 ROOT_PATH + APP_KEY + group 注册中心子目录列表
    List<String> serviceList = zkClient.getChildren(groupPath);
    if (CollectionUtils.isEmpty(serviceList)) {
        continue;
    }
    for (String service : serviceList) {
        //获取 ROOT_PATH + APP_KEY + group +service 注册中心子目录列表
        String servicePath = groupPath + "/" + service;
        List<String> serviceTypes = zkClient.getChildren(servicePath);
        if (CollectionUtils.isEmpty(serviceTypes)) {
            continue;
        }
    }
}

```

```

    }
    for (String serviceType : serviceTypes) {
        if (StringUtils.equals(serviceType, PROVIDER_TYPE)) {
            //获取 ROOT_PATH + APP_KEY + group +service+serviceType
            String providerPath = servicePath + "/" + serviceType;
            List<String> providers = zkClient.getChildren(providerPath);
            if (CollectionUtils.isEmpty(providers)) {
                continue;
            }

            //获取服务提供者信息
            for (String provider : providers) {
                String[] providerNodeArr =
                StringUtils.split(provider, "|");

                ProviderService providerService = new
                ProviderService();

                providerService.setAppKey(appKey);
                providerService.setGroupName(group);
                providerService.setServerIp(providerNodeArr[0]);
                providerService.setServerPort(Integer.parseInt
                (providerNodeArr[1]));

                providerService.setWeight(Integer.parseInt
                (providerNodeArr[2]));

                providerService.setWorkerThreads
                (Integer.parseInt(providerNodeArr[3]));

                providerServices.add(providerService);
            }
        } else if (StringUtils.equals(serviceType, INVOKER_TYPE))
        {

```


注册中心子目录列表

```

        //获取 ROOT_PATH + APP_KEY + group +service+serviceType
String invokerPath = servicePath + "/" + serviceType;
List<String>          invokers          =
zkClient.getChildren(invokerPath);

        if (CollectionUtils.isEmpty(invokers)) {
            continue;
        }

        //获取服务消费者信息
        for (String invoker : invokers) {
            InvokerService      invokerService      =      new
InvokerService();

            invokerService.setRemoteAppKey(appKey);
            invokerService.setGroupName(group);
            invokerService.setInvokerIp(invoker);
            invokerServices.add(invokerService);
        }
    }
}

return Pair.of(providerServices, invokerServices);
}
}

```

调用测试代码:

```

public static void main(String[] args) {
    IRegisterCenter4Governance invoker = RegisterCenter.singleton();
    Pair<List<ProviderService>, List<InvokerService>> pair = invoker
.queryProvidersAndInvokers("com.ares.remoting.test.HelloService", "ares");
    List<ProviderService> providerServices = pair.getLeft();
}

```

```
List<InvokerService> invokerServices = pair.getRight();

System.out.println(providerServices.size() + " " + invokerServices.size());
}
```

8.2.3 服务调用链路跟踪实现原理

对每一次调用使用一个唯一标识将服务之间的调用串联起来，有助于排查线上问题。其实现原理：在服务调用发起方生成标识本次调用的唯一 ID，传递到服务提供方，然后将该 ID 使用 `ThreadLocal` 保存起来，在应用的业务代码里面使用拦截器统一从 `ThreadLocal` 中获取出来。如果是使用 `slf4j` 日志组件打印日志，可以存入 `MDC` 工具类，然后在 `log4j.xml` 或者其他类型的日志配置（如 `log4j2.xml`、`logback.xml` 等）中配置对应的占位符，附加在日志记录的头部，从而可以实现唯一 ID 的自动输出，使用唯一 ID 将一次调用的日志串联起来。这一点本书未做实现，请读者自行完成。

8.3 本章小结

本章详细介绍了服务治理的概念及具体实施的思路。同时对服务治理对于分布式服务框架的重要性做了分析说明，并给出了部分服务治理实现的具体代码与实现原理。

附录 A

如何配置运行本书完成的分布式服务框架

A.1 环境准备

安装 JDK7，安装 ZooKeeper，并在运行分布式服务框架之前，启动 ZooKeeper 服务。

A.2 工程结构

整个分布式服务框架工程结构如图 A-1 所示，下面就每个包实现的功能做一个简单的介绍，有利于读者快速阅读源码。

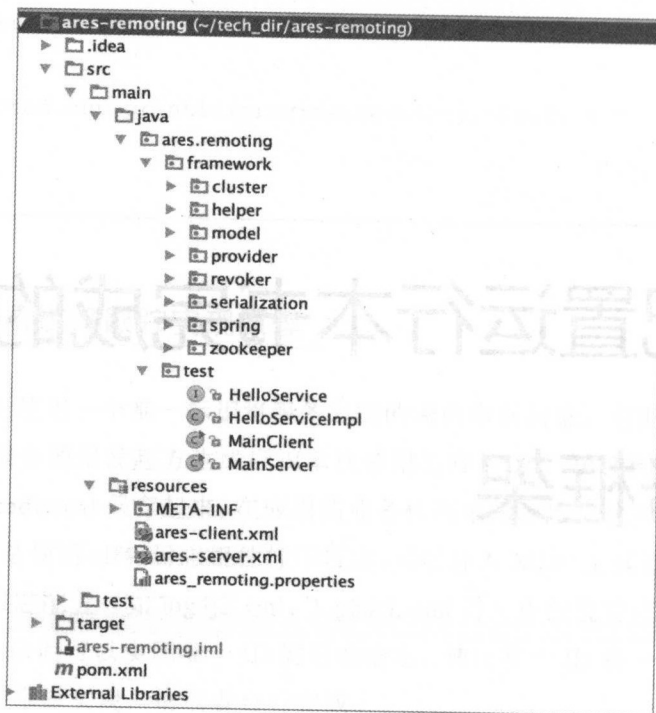


图 A-1 分布式服务框架工程结构

- ◎ ares.remoting.framework.cluster: 软负载实现。
- ◎ ares.remoting.framework.helper: 工具帮助类。
- ◎ ares.remoting.framework.model: 框架相关领域模型。
- ◎ ares.remoting.framework.provider: 服务发布实现。
- ◎ ares.remoting.framework.revoker: 服务引入实现。
- ◎ ares.remoting.framework.serialization: 数据序列化实现。
- ◎ ares.remoting.framework.spring: 服务发布/引入定制标签实现。
- ◎ ares.remoting.framework.zookeeper: 服务注册中心实现。

A.3 如何运行

以简单服务 `HelloService` 为例，说明如何进行服务的发布与引入运行。

接口 `HelloService` 代码如下：

```
package ares.remoting.test;

public interface HelloService {

    public String sayHello(String somebody);

}
```

实现类 `HelloServiceImpl`：

```
package ares.remoting.test;

public class HelloServiceImpl implements HelloService {

    @Override

    public String sayHello(String somebody) {

        return "hello " + somebody + "!";

    }

}
```

`ares-server.xml` 配置服务的发布：

```
<!-- 发布远程服务 -->

<bean

id="helloService" class="ares.remoting.test>HelloServiceImpl"/>

<AresServer:service id="helloServiceRegister"

    interface="ares.remoting.test>HelloService"

    ref="helloService"

    groupName="default"

    weight="2"

    appKey="ares"
```

```
workerThreads="100"
serverPort="8081"
timeout="600"/>
```

ares-client.xml 配置远程服务的引入：

```
<!-- 引入远程服务 -->
<AresClient:reference id="remoteHelloService"
    interface="ares.remoting.test.HelloService"
    clusterStrategy="WeightRandom"
    remoteAppKey="ares"
    groupName="default"
    timeout="600"/>
```

ares_remoting.properties 用来配置 ZooKeeper 相关的参数与序列化方案，例如：

```
zk_service=localhost:2181
zk_sessionTimeout=1000
zk_connectionTimeout=1000
channel_connect_size=15
#已支持
#DefaultJavaSerializer,HessianSerializer,JSONSerializer,
#ProtoStuffSerializer,XmlSerializer,MarshallingSerializer
#暂不支持 AvroSerializer,ProtocolBufferSerializer,ThriftSerializer
serialize_type=ProtoStuffSerializer
```

首先运行 MainServer 中的 main，将服务启动起来：

```
public class MainServer {

    public static void main(String[] args) throws Exception {
        //发布服务
        final ClassPathXmlApplicationContext context =
        new ClassPathXmlApplicationContext("ares-server.xml");
```

```
System.out.println(" 服务发布完成");
```

```
}
```

```
}
```

再运行 MainClient，引入远程服务，实现服务的调用：

```
public class MainClient {
```

```
    public static void main(String[] args) throws Exception {
```

```
        //引入远程服务
```

```
        final ClassPathXmlApplicationContext context =
```

```
new ClassPathXmlApplicationContext("ares-client.xml");
```

```
        //获取远程服务
```

```
        final HelloService helloService = (HelloService) context.getBean  
("remoteHelloService");
```

```
        //第一次调用等待初始化完成
```

```
        Thread.sleep(2000);
```

```
        //调用服务并打印结果
```

```
        for (int i = 0; i < 1000; i++) {
```

```
            String result = helloService.sayHello("liyebing");
```

```
            System.out.println(result);
```

```
        }
```

```
        //关闭 JVM
```

```
        System.exit(0);
```

```
    }
```

```
}
```

运行结果如图 A-2 所示。

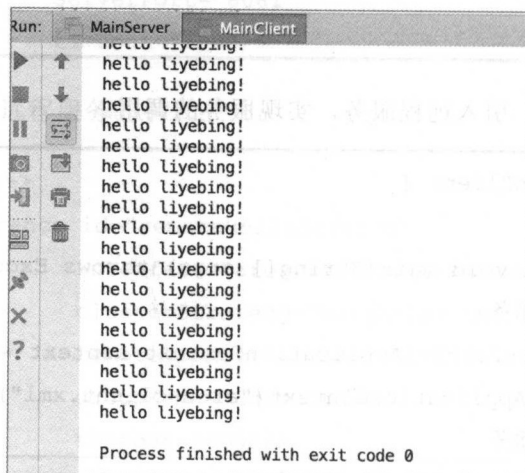
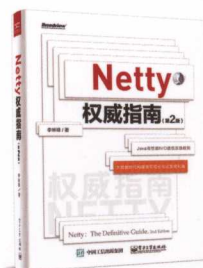


图 A-2 运行结果

好书力荐



反馈意见或投稿，联系编辑：

邮箱：dongying@phei.com.cn

微信：yingzidd

专家力荐

随着微服务的流行，支撑微服务的分布式服务框架成为大家优先发展的重点。服务化的基本原理掌握起来相对比较容易，但是要从零构建一个分布式服务框架却比较困难，涉及通信、线程并发、服务调度等。本书详细讲解了构建服务框架所需的各种技术及框架选型，手把手教初学者搭建一套完整的分布式服务框架。如果你想掌握分布式服务框架底层的技术细节，自己开发一套适合本公司和领域的服务框架，本书是一个非常不错的选择！

——华为架构师、《分布式服务框架原理与实践》《Netty权威指南》作者 李林锋

简洁明了的文字把技术介绍得深入浅出，实战操练的代码透漏出作者身经百战的深厚功力，通过本书不仅可以快速全面地了解 and 上手分布式服务框架开发技术，还能通过技术原理和内部实现的分析介绍，提升我们对技术的理解和洞察力。

——猫眼电影技术负责人 陈清阳

基于服务调用相关知识，业界已产出了不少技术书籍。本书汲取各家之精华，不同技术层级的人通过不同的章节都能获得极大的收获。更为难能可贵的是，作者是一个长期战斗在工程一线，一步一步成长起来的美团-大众点评技术专家，设计过大量的工程框架并主导开发实现，经受住了多种复杂业务形态的考验。凭借着作者本人多年的积累，凝聚出了书中丰富的示例和经验总结。通过这些示例和经验总结，内容逐渐深入，全方位地揭示了服务间通信的各项技术关键点，能有效地帮助读者从新手成长为专家。

——美团点评高级技术专家（原美团点评技术委员会委员） 黄波

本书比较全面地介绍了分布式系统开发的各方面知识，循序渐进，实例代码比较多，非常适合初入分布式开发并且有Java开发经验的人员参考学习。

——香格里拉酒店集团副总裁（原去哪儿网高级系统架构师） 孙立



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



责任编辑：董 英
封面设计：李 玲

上架建议：计算机 / 架构设计

ISBN 978-7-121-31959-4



9 787121 319594 >

定价：79.00元